NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

HYPER-NPSNET:EMBEDDED MULTIMEDIA IN A 3D VIRTUAL WORLD

by

Charles P. Lombardo

September 1993

Thesis Advisor:	Dr. Michael J. ZYDA
Thesis Co-Advisor:	LCDR John Falby, USN

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	September 1993	Master's Thesis

4. TITLE AND SUBTITLE
HYPER-NPSNET:EMBEDDED MULTIMEDIA IN A 3D VIRTUAL WORLD

5. FUNDING NUMBERS

6. AUTHOR(S)
Lombardo, Charles P.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/ MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/ MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*
As virtual world systems continue to evolve, the need exists to embed multimedia information into the world so users can query objects for additional information while maintaining frame rates greater than 15 frames per second. The need also exists for software tools to aid in the creation of multimedia documents intended for virtual worlds. This thesis addresses the problems of how to attach/query multimedia information to/from 3D locations in a virtual world and the design of a Graphical User Interface (GUI) to facilitate the creation of multimedia documents. The method chosen is to attach the multimedia information files to fixed 3D locations called Anchors. The anchors can be queried by the user and the multimedia information retrieved. Through the same interface, users can create multimedia documents by creating and/or editing anchor properties. The approach used differs from previous work in that navigation through the virtual world is unconstrained and a variety of information types may be attached to a single anchor. With video running and fully interactive navigation underway, the implementation presented gives rendering performance greater than15 frames per second for high-end graphics workstations.

14. SUBJECT TERMS
Virtual Worlds, Virtual Environments, Hypermedia, Multimedia, User Interface

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	SAR

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)

Approved for public release; distribution is unlimited

HYPER-NPSNET: EMBEDDED MULTIMEDIA IN A 3D VIRTUAL WORLD

by
Charles P. Lombardo
GS12, Civilian DON
MS, University of Washington, Seattle, 1988

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 1993

Author: ________________________________
Charles P. Lombardo

Approved By: ________________________________
Dr. Michael J. ZYDA, Thesis Advisor

____
LCDR John S. Falby, Thesis Co-Advisor

____
Ted Lewis, Chairman,
Department of Computer Science

ABSTRACT

As virtual world systems continue to evolve, the need exists to embed multimedia information into the world so users can query objects for additional information while maintaining frame rates greater than 15 frames per second. The need also exists for software tools to aid in the creation of multimedia documents intended for virtual worlds. This thesis addresses the problems of how to attach/query multimedia information to/from 3D locations in a virtual world and the design of a Graphical User Interface (GUI) to facilitate the creation of multimedia documents. The method chosen is to attach the multimedia information files to fixed 3D locations called Anchors. The anchors can be queried by the user and the multimedia information retrieved. Through the same interface, users can create multimedia documents by creating and/or editing anchor properties. The approach used differs from previous work in that navigation through the virtual world is unconstrained and a variety of information types may be attached to a single anchor. With video running and fully interactive navigation underway, the implementation presented gives rendering performance greater than15 frames per second for high-end graphics workstations.

TABLE OF CONTENTS

I. INTRODUCTION

As the evolution of real-time interactive 3D computer graphics continues, the need to manage and administer vast amounts of information will continue as well. As new frontiers are discovered, there will certainly be changes associated with the storage and retrieval of information. One of these changes that has already taken hold is the concept of multimedia. Multimedia consists of video and audio information in addition to the more usual text and picture (i.e. graphics) information. Across many platforms, but mainly PCs and Macs, the development of tools to aid in the creation of multimedia databases is quite active. One area that has not been so populated with advancements is the area of virtual environments with embedded multimedia. It is the focus of this work to embed multimedia capabilities into a real-time interactive 3D virtual environment and develop the necessary user interface and underlying data structures to make it all work; this is termed hypermedia. The system is called Hyper-NPSNET and is a prototype of a future version of the popular battlefield simulator NPSNET developed at the Naval Postgraduate School (NPS) [Zyda91][Zyda92].

A. THE IMPORTANCE OF HYPERMEDIA

There is a continuing need to manage not only data but the structure of data in the computer dominated environments where an ever increasing number of us work and live. The vast amounts of data are no longer simply binary numbers that must be crunched, but highly detailed photographs, diagrams, electronic circuit layouts, video used in education and in the media, endless news articles broadcast across vast computer networks, etc., etc. We not only want to know the names of pictures and videos, we want to quickly scan the available titles to make decisions on what may be relevant or deserving of further attention. We would like the capability to scan the titles of every research paper or movie that has a particular phrase in the title. For identification purposes, we may want to see a computer generated picture of all employees that work for the computer science department and, after selecting one or more, to see their performance reviews for the last six years. The point here is that different forms of data exist and the need to develop methods of dealing with them is one of the top computer software priorities for the future.

B. THE NEED FOR HYPERMEDIA IN VIRTUAL ENVIRONMENTS

As the whole Virtual Reality (VR) obsession explodes, many examples come to mind of beneficial uses of VR. Many of these fall under the category of tutoring systems. These may include the training of surgeons using virtual operating rooms, or the training of military personnel in the use of weapons by submersing individuals in a virtual war time environment. Examples like these go on and on and its clear that these kinds of systems can only benefit from the addition of multimedia capabilities. Consider the intern surgeon learning a procedure for cesarean section. Just using the virtual operating table may save lives or allow the intern to learn the procedure quickly, but if the intern was also given the capability to replay the operation or to retrieve video clips on the correct execution of some aspect of the procedure, the overall benefit would be much greater. The system could monitor the mistakes made and offer additional information that the intern should consider in applying a particular technique. In addition to video clips, the intern may request certain statistics during the operation. These may include the probabilities of complications from any of a range of circumstances for this particular patient. The ability to query VR systems for a range of information types on demand will allow a more complete immersion into the virtual world.

C. HYPER-NPSNET INTRODUCTION

Within Hyper-NPSNET, there is the notion of the information anchor. These anchors are repositories for a variety of multimedia information. The system can have a large number of anchors, each with hooks into video, audio, textual and graphics media. The user navigates through the system and chooses, either directly or through proximity to an anchor, the multimedia information to view or hear.

In the current system, the user interface consists of multiple Motif panels to administer the information anchors and a Silicon Graphics Incorporated (SGI) GLX Widget for rendering the virtual world. A 3D terrain database is read and used with texture information to create the ground of the virtual world. A multitude of buildings, trees, rocks, telephone poles and other miscellaneous objects populate the world. The end user typically loads an anchor database through the use of pull-down menus and pop-up windows (see “LOADING AND SAVING HYPERMEDIA DATABASES” in the Appendix). This database is created through the “Authoring” capabilities of the system (see “AUTHORING WITH HYPER-NPSNET” in Section V.). Typically the user chooses to have the anchors visible at all times. This is a visual cue of where the information anchors are attached to the world. The user can trigger any of the available multimedia information by first selecting an anchor and then pressing one of the four buttons: Audio, Video, Graphics or Text on the Hyper-NPSNET main control panel. To select an anchor, the user either selects it with the mouse directly in the 3D world, or chooses it off a list of available anchors displayed in the main control panel. Upon selection of an anchor, the main control panel displays the current anchor name, type and coordinates. In addition, the current anchor is highlighted on the scrolling list of all available anchors. If the user selects the anchor off of the list, then the viewing point in the rendering window is transported to the location of the anchor in the 3D world. This is referred to as an instant aspect change. No matter how the anchor is selected, the user knows what kind of multimedia information is available for this anchor by which of the Audio, Video, Graphics and Text buttons are sensitive.

To gain access to all the anchors in the system, the user navigates through the world using either a Spaceball, Ascension Bird or standard 2D mouse. Through trials, it was determined that the most intuitive device and a device found on virtually every workstation was the 2D mouse. The Spaceball made it easy to move around, but difficult to pick anchors within the 3D world. The Ascension Bird introduced a disconcerting shakiness to the display that could not be overcome.

The user has a number of preferences that can be set through the use of the "Preferences" pop-up panel. Here the user can specify whether to fly around the world or drive on the terrain. If driving, the user steers left or right by moving the cursor left or right outside a small control square in the middle of the rendering window. To speed up, move the mouse up, to slow down or go backwards, move the mouse down. If flying, the heading is set with the left-right motion of the mouse, and the pitch is set using the up-down motion of the mouse. This leaves the speed to be controlled by some other means, so the user can specify the flight speed in the preferences panel. The user can also specify whether local anchors only are to be displayed. If local anchors only are being displayed, the user can specify the range that defines what local is. The default value is 300 meters (the current terrain is 2 km by 2 km) meaning that only anchors within 300 meters of the current position of the user are displayed. The last thing the user can specify in the preferences pop-up panel is whether anchor information is displayed automatically as the user gets close to one of the anchors. If chosen, the user can specify the range used to trigger the multimedia information and can choose what information is automatically displayed. The default is Anchor Auto View off with a range of 20 meters and Audio media tagged. So if the user sets Anchor Auto View on and leaves other default values alone, then anchor audio tracks will play anytime the user gets within 20 meters of an anchor. This is known as *Audio Landmines*.

Hyper-NPSNET can be used as an authoring tool for hypermedia. To build a new hypermedia database, the system is brought up without loading an anchor database. The user then creates all the anchors and attaches all the multimedia using the Anchor Editor Tool. For existing anchors, the editor is used to change any of the values for the anchor:

name, type, coordinates, orientation, audio track filename, video filename, graphics filename and text filename. For new anchors, the user simply enters all the information about the new anchor and saves it to the system. An anchor will appear in the 3D location specified by the coordinates and will have all of the multimedia information available for viewing or hearing. Note that the audio, video, graphics and text files have already been created so the role of Hyper-NPSNET is a multimedia player not recorder, but it could easily be a recorder through the addition of a video cam and some software (see “FUTURE WORK” on page 56).

D. THESIS ORGANIZATION

In Chapter II, a survey of appropriate previous work is presented. This is followed by a discussion of the system requirements necessary to implement Hyper-NPSNET in Chapter III. Chapter IV presents the underlying data structures for the hypersystem and graphical user interface. How Hyper-NPSNET is used as an authoring tool is covered in Chapter V, and the results of this work are in Chapter VI. Recommendations for further work are in Chapter VII. This is followed by the appendix, containing the Hyper-NPSNET user manual. The appendix is followed by the list of references.

II. SURVEY OF PREVIOUS WORK

In the last few years, a number of multimedia systems have been developed and marketed. The majority of these are designed to be run on PC class machines and therefore cannot sport a fully interactive real time virtual world interface. The question: When would a fully interactive 3D multimedia system be required? This question is not easily answered, but if the capability and the cost of the hardware were not at issue, there would be many more such systems available than there currently are.

It is difficult to find information on or hear of these kinds of systems. The best sources are the journals, but these articles seldom give any substantial clues to the underlying design. Rather they describe the application and leave the rest to the imagination. The following sections present summaries of three real-time interactive hyper-media systems. The first is an interactive walk through of a virtual museum built with an object-oriented toolkit designed to aid in the construction of distributed multimedia applications. The second is a fully interactive 3D information system, with the third being a multimedia publication and document structuring system. First, let's define what hypermedia is.

A. HYPERMEDIA DEFINITION

Hypermedia is defined [Halasz88] as:

> Hypermedia is a style of building systems for information *representation* and *management* around a network of multi-media nodes connected together by typed links. Such systems have recently become quite popular due to their potential for aiding in the organization and manipulation of irregularly structured information in applications ranging from legal research to software engineering.

Nielsen defines Hypertext as the nonsequential access of text, and Hypermedia as the nonsequential access of different media including text [Nielsen90]. This definition is rather vague and differs from Halasz's definition in so much that Halasz suggests that the interface should represent the structure of the system. By this he means the network of the inter-connecting links and nodes within the system.

The basic physical definition of what a hypertext or hypermedia system is can be explained using a simple example about a hypertext document from [Nielsen90] (see Figure 1).

> Assume that you start by reading the piece of text marked **A.** Instead of a single next place to go, this hypertext structure has three options for the reader: Go to **B**, **D**, or **E**. Assuming that you decide to go to **B**, you can then decide to go to **C** or to **E**, and from **E** you can go to **D**. Since it was also possible for your to go directly from **A** to **D**, this example shows that there may be several different paths that connect two elements in a hypertext structure.

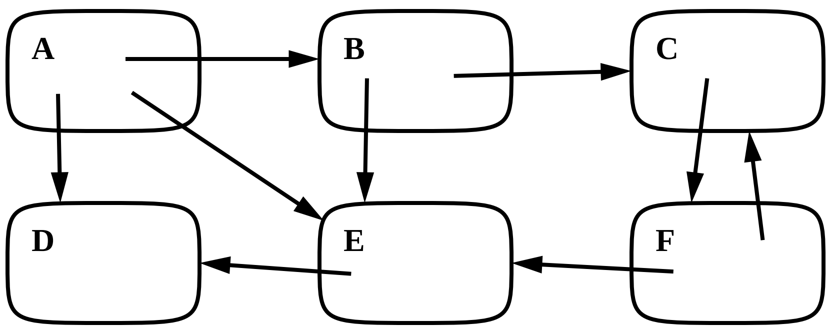

Figure 1: Simplified view of a small hypertext structure having six nodes and nine links

B. THE VIRTUAL MUSEUM

The *Virtual Museum* is a walk through of a simulated museum with displayed artifacts that can be interacted with by the user [de Mey93]. The main focus of this work is the design of an object oriented "Multimedia Component Kit". The components are used to assemble multimedia applications. The Virtual Museum is an example application implemented using media objects from the component kit. These objects are responsible for navigation, rendering and media stream input/output, among others.

The user interacts with the museum by navigation and selection of artifacts. The types of artifacts include still pictures and animated video sequences projected onto different surfaces. The software also includes a sound server to maintain the audio feedback to the user. As the user navigates the museum, various artifacts are visible in a reduced resolution rendering. At a certain point when the viewer is close enough, the artifact is displayed in its maximum resolution. If the artifact involves video, the animation starts. Various aural cues are triggered as the user moves about.

Applications built using the component toolkit (including the virtual museum) are constructed using visual composition. This means that once media objects are defined, like the navigation object, they can be connected together interactively to form applications. Actually, iconic representations of the objects are connected together using the mouse. The objects have interfaces through which they are connected to other objects. These interfaces are also shown on the icon. Using these *connection points* the user assembles the application.

C. THE INFORMATION VISUALIZER AND RELATED STUDIES

The *Information Visualizer* is a real time interactive 3D information displaying front end to an information storage system developed at the Xerox Palo Alto Research Center (Xerox PARC) [Card91]. The main focus of the system is 1) the use of 3D/Rooms for increasing the immediate storage capacity, 2) the user interface to couple the user to the information agents, and 3) 3D visualization for real time interaction with the data.

The overall concern of the Information Visualizer is to allow access to more information more quickly than otherwise possible. As pointed out in [Card91], there is a cost structure associated with the retrieval of information. The analogy is made to an office and the different types of information readily available, like file cabinets and computer-based information retrieval systems. If the office layout is designed well, the costs of accessing the information will be minimal. For instance, a Rolodex is kept on the desk, so the cost of retrieving phone numbers is low. This makes sense because the Rolodex is used frequently. For less frequently used storage media like the filing cabinet, the information is in a higher cost structure. At the highest cost might be information not available in the office at all, but at the local public library. The desk side information is considered *Immediate Storage*, the file cabinet is considered *Secondary Storage* and the library is *Tertiary Storage.*

Once the cost structure of information is realized, then the cost of assimilation becomes important. The Information Visualizer attempts to minimize these costs by utilizing *Information Workspaces*. Computer screens play the roll of workspaces. Immediate storage information is displayed on the workspace as menus, windows and icons. Secondary storage is represented as virtual rooms each having multiple workspaces. The 3D/Rooms can be navigated, by the user, to explore the information system. The users orientation can be changed and objects can be manipulated with the mouse. Using navigation, for example, the user can zoom in to show more detail anywhere in the room.

In related studies at Xerox PARC, information is displayed using animated real time 3D computer systems. In one of these studies, the concept of a *Cone Tree* is presented [Robertson91]. Cone trees are hierarchies laid out in three dimensions. In the study, a cone tree was used to develop a UNIX file browser. The cone tree represents the directory structure, with each node representing a directory in the file system. The display shows the directory structure as a cone facing down. The root directory is at the top. The user can select any directory using the mouse. Upon selection, the entire directory structure rotates until the chosen directory is in front of the user. It was determined that an immediate change of the display was disorienting to the user, so animation was incorporated to allow the user to perceive the change.

The cone tree paradigm works well with any hierarchical information system, but for vast systems with little hierarchy, the *Perspective Wall* supports efficient use of space and time [Mackinlay91]. The Perspective Wall allows the visualization of large linear or nearly linear data sets. The display shows a wall directly in front of the user where information is presented. On either side of the front wall is another wall angled back with the appearance of being folded. The folding metaphor is used to distort the 2D layout into a 3D visualization. The center wall is for viewing detail and the side walls are for viewing context. The user can select any feature on any wall with the mouse. Once selected, the wall moves that item to the center panel with a smooth animation. As in the Cone Trees, this animation helps the user perceive the change in the displayed system.

D. MEDIAVIEW

MediaView is an editable multimedia publication system[Phillips91]. Even though MediaView does not incorporate any Virtual World paradigms, it does allow for real time audio, video and graphics to be imbedded into documentation. The user interface is based on the what-you-see-is-what-you-get (WYSIWYG) word processor metaphor. Text and all multimedia objects are subject to the select/cut/copy/paste operations of most modern word processors. This makes manipulating existing multimedia very easy and allows for the creation of multimedia documents by nonspecialists.

The potential applications for MediaView are numerous. Imagine fully interactive textbooks that would allow students to query the system for additional information about any number of topics in a nonsequential manner. In fact, selected chapters of *Computer Graphics: Principles and Practice* have been transformed into MediaView documents, making it possible to animate algorithms, explore mathematical expressions, and view 3D databases. Additional potential lies in such areas as interactive scientific visualization for Science and Engineering, and digital patient records for Medicine, where photographs or medical data like EKG results can be presented. Training systems can incorporate video segments in a multimedia shop manual.

E. WHAT IS SO SPECIAL ABOUT HYPER-NPSNET?

Hyper-NPSNET is different from the examples above and from all other such systems that the author is aware partly because of the unconstrained virtual world that the hypermedia links and nodes are embedded in. Within Hyper-NPSNET the user can move through-out the virtual environment unconstrained[1]. This differs from the Information Visualizer because in that system the user can only “wonder” where there is data. In Hyper-NPSNET, the user can navigate over barren terrain even if there are no information nodes present. The user may be training to drive a tank while looking for enemy vehicles or looking for landmarks. Information nodes are not everywhere, they must be found. Another unique feature of Hyper-NPSNET is the ability to attach up to four distinct types of media information to one physical location. This differs from the Virtual Museum in that only one artifact is placed at any location.

Upon approaching an anchor, the user can query it for additional information. These queries can be for information about the current anchor’s location and orientation in the world or for a playback of some audio or video track that has been attached to the anchor by the author of the data set. The user can also select any anchor off of a list of all nodes in the system. As these nodes are visited, links are established, allowing the user to “Back out” in the reverse order of the initial visits. These features make Hyper-NPSNET unique from the previous work cited in this chapter.

1. That is until the user hits the edge of the world. Currently the world is on a 2km terrain. Upon reaching the edge, the user must turn back in to interact with anything meaningful.

F. TERMINOLOGY CONVENTIONS

The terminology used for the remainder of this thesis differs slightly from that established by [Nielsen90]. A 3D location in the virtual world that has an identity and the capability to attach audio, video, graphics or textual information is called an *Anchor*. Each anchor can have associated with it up to four *nodes* of distinct types. Each node is represented by the audio, video, graphics or text file attached to it. The *links* are established between anchors, and between an anchor and it’s associated information nodes, but not between information nodes. So after visiting Zydaville 1, if the user then visited the Command Post, a link is established between them, allowing the user to revisit Zydaville 1 by backing up with the *Back* button on the Hyper-NPSNET main control panel. All the anchors, links and information nodes comprise the *Hypersystem*.

III. HYPER-NPSNET SYSTEM REQUIREMENTS

Hyper-NPSNET is written in C++ and runs on commercially available SGI IRIS workstations in all its incarnations. The hardware and software requirements are discussed in more detail in the following sections.

A. SOFTWARE REQUIREMENTS

The software for Hyper-NPSNET consists of 16 C++ classes each with a specification file (i.e. .H file) and an implementation file (i.e. the .C file). In addition to the classes, there are 13 C++ .C files, 9 .H files and an assortment of other miscellaneous files. The total word count output looks like:

```
 157     489    4269      Anchor.C
  67     300    2317      Anchor.H
 626    2078   19674      Control.C
  87     376    3361      Control.H
1207    4572   43028      EditAnchor.C
  76     344    3043      EditAnchor.H
 353    1401   11497      Face.C
  41     142    1154      Face.H
 485    1530   14218      FileInfo.C
  53     173    1561      FileInfo.H
   9      16     247      Fileops.C
  71     210    2214      Global.C
  93     529    4310      Global.H
 210     670    5863      Graphic.C
  43     134    1178      Graphic.H
 101     270    3023      Hnode.C
  59     199    1764      Hnode.H
  16      43     487      Hstack.C
  23      70     675      Hstack.H
 237     639    6134      Hsystem.C
  57     223    1886      Hsystem.H
 299    1101   10007      Lister.C
  51     169    1468      Lister.H
 923    3215   30484      MenuBar.C
  78     250    2276      MenuBar.H
  29      92     809      MessBox.C
  25      44     455      MessBox.H
 150     410    4490      Panel.C
```

Figure 2: File sizes in lines, words and characters for Hyper-NPSNET software

53	144	1315	Panel.H
1044	3722	34862	Preferences.C
69	309	2734	Preferences.H
156	651	4947	TextBox.C
29	61	589	TextBox.H
832	2592	24135	XInput.h
75	251	2102	butt.H
316	858	6122	cube.C
59	144	1327	cube.H
90	229	1768	disdefs.h
8	45	318	externs.h
119	510	3367	getsgi.C
1054	3750	29978	glDraw.C
112	416	3372	glDraw.H
70	217	1860	hyper.C
50	136	1160	hyper.H
86	328	2446	image.H
86	300	1917	image_types.H
365	1654	12061	io.C
23	64	486	main.C
20	47	372	materials.h
10	41	410	objectsDraw.C
76	194	2012	rdobj_funcs.h
139	389	3432	readfiles.c
64	182	1861	shapes.C
7	11	124	shapes.H
82	236	1736	simnet.h
279	1074	9101	spaceball.C
73	251	2041	spaceball.H
198	888	7015	stationaryObjects.C
60	209	1920	stationaryObjects.H
36	115	463	test.C
208	755	6321	unite.C
25	72	777	unite.H
67	195	1890	utils.C
398	1489	10611	viewbounds.C
12064	42218	368844	total

Figure 2: File sizes in lines, words and characters for Hyper-NPSNET software (Continued)

Notice in the last line of Figure 2 above, the total number of lines of code for Hyper-NPSNET is about 12000. This does not include any of the supporting code that is used to define and display objects or that for the displaying of image files. For defining and displaying the 3D objects in the world, the NPS Graphics Description Language

(NPSGDL) system is used. NPSGDL was written at NPS and is a high level language for the specification and manipulation of 3D objects [Wilson92]. For the display of image files, a package also developed at NPS called NPSImage is used.

All the C++ code is AT&T C++ 3.0 compliant. There is some miscellaneous C code written in ANSI C 3.1. The complete user interface is written in the native Motif on IRIX Version 4.0.5. All of the above comprise the minimum requirement for porting this software to another platform. Note that the rendering is done in a GLX Widget that allows SGI Graphics Library (GL) rendering in a Motif widget.

B. HARDWARE REQUIREMENTS

The hardware requirements for Hyper-NPSNET parallel the software requirements as discussed above. A specific hardware requirement is audio capability. This coupled with the GL rendering require the code be run on an SGI IRIS platform. Beyond this, there is the consideration of disk space, memory and input devices.

1. Hard Disk Drive Capacity

Currently the audio and video formats used for the hypermedia are in a relatively uncompressed format. Typical movieplayer video files used by Hyper-NPSNET range from 700KB to 23MB in size. Yes that IS 23,000,000 bytes. The audio files range from 300KB to 2MB in size. This implies that for a hypermedia database consisting of 50 information anchors with unique audio and video links, the required disk space above and beyond the operating system and all normal user requirements is about 550MB. This assumes an average video size of 10MB and an average audio size of 1MB; rather conservative estimates. This doesn’t take into account the size of the terrain database, nor the image or text storage requirements. As compression techniques improve and become more widespread, these number will certainly change. A near term option that is being looked at for a future version of Hyper-NPSNET uses the Cosmo compress option.

2. Memory

Other than disk space, the amount of memory local to a machine plays an important role in how quickly and smoothly the audio and video clips are played. Typical memory requirements range from a minimum of 32MB to over 100MB. Since Hyper-NPSNET has only been run on SGI machines, it is not clear how well the system would run on a machine that typically doesn't deal in these kinds of numbers.

3. Input Devices

A number of input devices were used throughout the evolution of Hyper-NPSNET. All of the input devices to be discussed used the *eyeball in hand* metaphor [Robertson91]. The first to be tried was the 6 degree of freedom Spaceball. For users familiar with the spaceball, navigation is quite intuitive. The ball is grasped in the hand and is used to manipulate viewpoint motion. The major drawback with the spaceball is the difficulty in using it as a pick device. In order to select anchors in the 3D world, the cursor must be manipulated so as to orient it on top of an information anchor. This was near to impossible to do in a coordinated way with the spaceball.

The second device that was used was the *Ascension Bird*. The Ascension Bird is a 6 degree of freedom mouse, that is held by the user and allowed to move not only on a typical 2D surface but also up and down. The hardware includes a transmitter that generates a strong magnetic field, and a receiver to sense the field. As the Bird is moving in 3D space, the varying magnetic field is converted back to 3D coordinates. The problem we had with the Ascension Bird was an annoying jitter in the graphical display of the virtual world that could not be overcome. This jitter at a minimum disoriented the user and made movement and selection difficult.

The final method of navigation uses a standard 2D mouse. Whenever the user wants to move forward, the left mouse button is pressed and held down. If the user wants to move backward, then the right mouse button is used instead. While either button is pressed, a red square appears in the middle of the screen. To turn, the user moves the mouse cursor outside of the red box. If the cursor is moved to the right, movement is to the right. If the cursor is moved to the left, then movement is to the left. The rate at which the turning occurs is proportional to the distance the cursor is moved away from the edge of the box. The elevation of the vehicle is changed in a similar fashion by moving the cursor either above or below the edges of the box. To select any objects off the screen, the user simply positions the cursor over the anchor and presses and releases the middle mouse button. This method is easy to use and utilizes an input device that is found on virtually every workstation.

IV. HYPER-NPSNET DATA STRUCTURES

At the heart of Hyper-NPSNET, lie the data structures that make it all possible. Since Hyper-NPSNET is written in C++, these data structures correspond to C++ classes. The discussion that follows covers the classes for the hypersystem as well as the classes for the Motif based GUI. Sample C++ code is presented.

A. HYPERSYSTEM CLASSES

There are three levels of C++ classes that make up the hypersystem. The hypersystem is defined as the fundamental data structures that hold individual node information and all the underlying links that enforce the association between anchors and information nodes. At the lowest level resides the HyperNode. The HyperNode is the basic information containing entity of the system. An example of a HyperNode is a reference to an audio file that contains an audio track. Above the HyperNode is the Anchor. The anchor contains ("has a") up to four HyperNodes that can represent the audio, video, graphic and text information associated with the anchor. The anchor is like a hub with different kinds of information packets attached. A collection of anchors represents the HyperSystem. It is through the HyperSystem level that individual anchors are created, modified or destroyed. A fourth class is implemented that contains system-wide global information. This is the Global class. This class is responsible for maintaining system state information. In the following four sections, these classes are discussed in detail.

1. Hypernode

As mentioned above, the HyperNode is the fundamental information entity of the system. The HyperNode Class declaration contains private variables for maintaining node information (see Figure 3).

.

```
class HyperNode {
  unsigned id;                          // a unique identifier
  unsigned type;                        // type of node
  char filename[80];                    // filename pointed to
  static unsigned id_generator;         // auto init to zero
public:
  HyperNode();
  HyperNode (const HyperNode&);
  char* getFilename();
  unsigned getId() {return id;}
  unsigned getType() {return type;}
  HyperNode& operator=(const HyperNode&);
  void resetIdGenerator() {id_generator = 0;}
  void setFilename(char*);
  void setType(unsigned t) {type = t;}
};
```

Figure 3: HyperNode Class Declaration

Included in the node information is the node's identification, `id`. The node id can be retrieved using the `getId()` access function, but cannot be set. The node id is a unique identifier generated automatically by the node constructor through the use of `id_generator` [Wilson92]. Also maintained is the node type. `type` describes what kind of information is held by the node. Currently there are six recognizable node types: NODE_UNKNOWN, NODE_GENERAL, NODE_AUDIO, NODE_VIDEO, NODE_GRAPHIC and NODE_TEXT. NODE_UNKNOWN and NODE_GENERAL are not currently implemented, but are defined for future use. The other node types are self explanatory. The node type can be retrieved using the `getType()` member function and set through use of the `setType()` member function. A node points to a file that contains either audio, video, graphics or textual data. The filename pointed to is maintained in `filename[80]`. The filename string can be set with `setFilename()` and retrieved with `getFilename()`.

2. Anchor

The anchors correspond to the abstract information containers. It is the bringing together of the information within the HyperNodes that makes the Anchor. Associated with any anchor, there can be audio, video, graphic or textual information attached. The user merely asks to see and/or hear the information and it is presented. To make this work, the anchor needs hooks in up to four HyperNodes. As shown in the Anchor declaration (see Figure 4), an instance of an anchor stores the HyperNode ids internally. This is done in the `audio`, `video`, `graphic` and `text` private variables. These are unsigned integers and merely hold the node id that was automatically generated by the node constructor (see "Hypernode" in Section IV.A.1.). In all four, the unsigned integer id can be retrieved and set with the appropriate access functions. For instance, to set or get the video node's id, `setVideo()` or `getVideo()` is used.

As mentioned earlier, the node ids are unique and this guarantees no collisions on the node level. This minimizes the amount of information necessary to uniquely identify the appropriate information. In addition to node info, the anchor must maintain information specific to itself. This includes it's own id, which is used later to identify the current anchor and facilitates retrieval of relevant information in a timely fashion. The anchor id, like the node id, can be retrieved with `getId()` but not set. The anchor type is retrieved with `getType`() and set with `setType()`. `type` represents the kind of information object we are dealing with. Currently only TERRAIN anchor types are implemented. TERRAIN anchors are attached to the terrain and once created cannot be moved. Additional types might include VEHICLE and TEMPORAL anchors. A vehicle anchor, then, is an information object attached to a potentially moving vehicle that carries information with it. The goal here is to have information available pertaining to the weapons systems or design capabilities of various vehicles that move around on the terrain and that may be engaged at different times throughout the simulation. Temporal anchors allow for the creation of anchors that exist in some kind of predetermined time space, for instance, having an anchor available for the duration of a particular ground engagement only. This anchor might

```
class Anchor {
  unsigned id;
  unsigned type;                     // The type of anchor
  char name[40];                     // anchor name
  float coords[3];                   // the coordinates of the anchor
  float orientation;                 // the orientation angle
  unsigned audio;                    // id of audio hypernode
  unsigned video;                    // id of video hypernode
  unsigned graphic;                  // id of graphic hypernode
  unsigned text;                     // id of text hypernode
  static unsigned id_generator;      // automatically init to zero
public:
  Anchor();
  Anchor(char *);
  Anchor(const Anchor&);
  Anchor(AnchorPtr);
  unsigned getAudio()                {return audio;}
  float* getCoords()                 {return coords;}
  unsigned getId()                   {return id;}
  unsigned getGraphic()              {return graphic;}
  char* getName()                    {return name;}
  float getOrientation()             {return orientation;}
  unsigned getText()                 {return text;}
  unsigned getType()                 {return type;}
  unsigned getVideo()                {return video;}
  void resetIdGenerator()            {id_generator = 0;};
  void setAudio(unsigned a)          {audio = a;}
  void setCoords(float, float, float);
  void setGraphic(unsigned g)        {graphic = g;}
  void setName(char *);
  void setOrientation (float o)      {orientation = o;}
  void setText(unsigned t)           {text = t;}
  void setType(unsigned t)           {type = t;}
  void setVideo(unsigned v)          {video = v;}
};
```

Figure 4: Anchor Class Declaration

maintain information relevant to enemy ground movement that is only useful during the engagement.

The anchor name is used to identify the anchor to the user. This name is displayed on the main control panel for Hyper-NPSNET (see “Main Panel” in Section IV.B.1.). The anchor name is retrieved with `getName()`. `getName()` returns the anchor name as a character string. The name can be set or changed with `setName()`. `setName()` takes a character string as an argument. The anchor name can be up to 40 characters in length.

Additional information that the anchor is responsible for deals with its current position and orientation. The current position and orientation are maintained in `coords` and `orientation` respectively. The current position is retrieved with `getCoords()`, which returns a pointer to three floating point values that represent the x, y, and z coordinates. The position can be set or changed with `setCoords()` that takes the three floats as arguments. The orientation is retrieved with `getOrientation()` and returns a floating point value that represents an angle between 0 and 360 degrees. The orientation can be set or changed with `setOrientation()` that takes the float as an argument.

3. HyperSystem

Once the anchors are designed, the information is assembled into a manageable object. This object is the HyperSystem. Through the HyperSystem class, all anchor and HyperNodes in the system are maintained (see Figure 5).

As can be seen in the class declaration, the HyperSystem object maintains information about the total number of anchors and the total number of nodes in the system. This is done through the `n_anchors` and `n_nodes` private variables. In addition to this, a list of pointers to all anchors and a separate list of pointers to all nodes is kept in `anchor_list` and `node_list`. This is done to facilitate rapid searching for information location and for checking on information duplication. Since the Hypersystem locates both anchors and nodes using a unique id, some means of decoding is necessary. The decoding is done through the private variables `anchor_table` and `node_table`. These are simple data structures that allow for rapidly locating the address of an anchor or node given its id (see Figure 6).

As seen in Figure 7, the member function `GetAnchorPtr()` does a linear search on the anchor table. Upon finding the input id, the function returns the associated address.

```
class HyperSystem {
  unsigned n_anchors;              // number of anchors in the sys
  unsigned n_nodes;                // total number of nodes
  List(AnchorPtr) anchor_list;     // list of all anchors
  List(HNode) node_list;           // list of all nodes
  AnchorTable anchor_table[10];    // anchor id to address cnvrsn
  NodeTable node_table[40];        // node id to address cnvrsn
public:
  HyperSystem();
  List(AnchorPtr)& AnchorList() {return anchor_list;}
  void clearAll();
  AnchorPtr CreateAnchor ();       // create and attach an anchor
  AnchorPtr CreateAnchor (AnchorPtr);
  HNode CreateNode ();             // create and attach a node
  HNode CreateNodeForAnchor (AnchorPtr, unsigned, char*);
  AnchorPtr GetAnchorPtr (unsigned);
  HNode getHNode (unsigned);
  unsigned NAnchors() {return n_anchors;}
  unsigned NNodes() {return n_nodes;}
  List(HNode)& NodeList() {return node_list;}
};
```

Figure 5: Hypersystem Class Declaration

```
// set up the tables necessary for associating id's
// with actual addresses
typedef struct anchortable {
  unsigned id;                     // the anchor id
  AnchorPtr address;               // the anchor address
} AnchorTable;

typedef struct nodetable {
  unsigned id;                     // the node id
  HNode address;                   // the node address
} NodeTable;
```

Figure 6: Structures used to decode Anchor and Node addresses

The member function `getHNode()` is virtually identical in function to that of `getAnchorPtr()`, but returns an address of a node instead.

```
AnchorPtr HyperSystem::GetAnchorPtr (unsigned id)
{
 AnchorPtr a_ptr;
 for (register i = 0; i < n_anchors; i++) {
   if(anchor_table[i].id == id) {
     a_ptr = anchor_table[i].address;
     break;
   }
 }
 return(a_ptr);
}

HNode HyperSystem::getHNode (unsigned id)
{
 HNode node;
 for (register i = 0; i < n_nodes; i++) {
   if(node_table[i].id == id) {
     node = node_table[i].address;
     break;
   }
 }
 return(node);}
```

Figure 7: HyperSystem Anchor and Node Rapid search

As an example, consider the user/programmer wanting to get or set the audio track information for the current anchor. To retrieve the current audio filename of the active anchor, the programmer makes a call similar to:

```
audio_filename =
GetHNode (GetAnchorPtr (current_anchor_id). getAudio()).getFilename()
```

The equivalent call to set the audio filename looks like:

```
GetHNode (GetAnchorPtr (current_anchor_id).getAudio()).set-
Filename(“new_file_name”).
```

4. Global Class

The responsibility of the Global class (see Figure 8) is to maintain information about the current state of the HyperSystem. `anchor_editor_open` is set when the anchor editing tool is displayed (see "Anchor Editor Panel" in Section IV.B.2.). `current_anchor_id` holds the current anchor id (see "Anchor" in Section IV.A.2.). `current_filename[80]` contains the hypermedia database filename that has been read into Hyper-NPSNET. `display_anchors` is set when the user sets the anchors visible. `display_texturing` is set when texturing is turned on. `display_visible_anchors` determines whether all anchors are visible or just ones that are within a certain distance from the user. In addition, the stack of visited anchors is kept in `hstack` to allow backing out of the anchors in reverse order of visitation (see "Main Panel" in Section IV.B.1.). This gives Hyper-NPSNET a hypertext-like capability. As anchors are visited, they are pushed onto the stack. To revisit the last anchor or anchors, they are popped off the stack. A pointer to the HyperSystem is also kept in `hsystem` to facilitate information queries. `save_status` keeps track of whether the system has been changed by the user and `texture_bound` reflects whether textures are currently bound by the program. The texture information is kept to allow the texturing of certain objects, like the terrain, without affecting objects that are not textured, like the information anchors.

Although most of the Global class member functions are self explanatory, the two panel functions deserve a comment. The idea of the Global class is that from any module in Hyper-NPSNET, certain state information is available. Since system state changes can be initiated from many different pieces, there must be a mechanism to inform the panel to update itself whenever a system state change occurs. This is done through the `getPanel()` access function. As shown below, the panel class itself can be updated with a call to its own member function `updateYourself()` (see "Main Panel" in Section IV.B.1.). So, from anywhere in the system, the state of the panel can be update by:

```
global->getPanel().updateYourself()
```

where `global` is an instance of the Global class.

```
class Global {

  unsigned anchor_editor_open;      // is anchor editor up or down
  unsigned current_anchor_id;       // the one thats highlighted
  char current_filename[80];        // set after opening a file
  unsigned display_anchors;         // are the anchors being shown
  unsigned display_texturing;       // to texture or not
  unsigned display_visible_anchors;// display only visible guys
  HStack hstack;                    // the stack of visited anchors
  HyperSystem hsystem;              // the hypersystem
  Panel *panel;                     // the panel interface
  unsigned save_status;             // i.e. is a save needed
  unsigned texture_bound;           // are textures currently bound

public:

  Global();
  char* getCurrentFilename() {return current_filename;}
  unsigned getCurrentAnchorID() {return current_anchor_id;}
  unsigned getDisplayAnchors() {return display_anchors;}
  unsigned getDisplayTexturing() {return display_texturing;}
  unsigned getDisplayVisibleAnchors()
                     {return display_visible_anchors;}
  unsigned getEditorDisplayState() {return anchor_editor_open;}
  HyperSystem& getHypersystem() {return hsystem;}
  HStack getHstack() {return hstack;}
  Panel *getPanel() {return panel;}
  unsigned getSaveStatus() {return save_status;}
  unsigned getTextureBound() {return texture_bound;}
  void setEditorClosed() {anchor_editor_open = EDITOR_CLOSED;}
  void setEditorOpen() {anchor_editor_open = EDITOR_OPEN;}
  void setCurrentFilename (char*);
  void setCurrentAnchorID (unsigned id) {current_anchor_id = id;}
  void setDisplayAnchors (unsigned a) {display_anchors = a;}
  void setDisplayTexturing (unsigned a) {display_texturing = a;}
  void setDisplayVisibleAnchors (unsigned a)
                     {display_visible_anchors = a;}
  void setPanel (Widget);
  void setSaveStatus (unsigned s) {save_status = s;}
  void setTextureBound (unsigned a) {texture_bound = a;}

};
```

Figure 8: Global Class Declaration

B. GRAPHICAL USER INTERFACE (GUI) CLASSES

The GUI for Hyper-NPSNET is perhaps the most complicated aspect of the program. As is the case with many pieces of software, the interface is the program. Part of the goal for this work revolved around an object oriented design paradigm, but another part was to make use of sophisticated toolkits for building the user interface. Since this was to be implemented on a Silicon Graphics workstation porting a native X Windows System, it seemed natural to pick Motif as the toolkit. This turned out to be a good choice although the learning curve for Motif is quite steep.

The user interface consists of the main control panel that the user interacts with quite frequently and a collection of pop-up dialog boxes that present themselves when appropriate. For the most part, these components are either a self contained C++ class or a collage of other C++ classes. All of the classes are responsible for updating themselves as the state of the simulation changes.

1. Main Panel

The main panel contains four C++ classes. The four classes defined are the MenuBar, Face, Control and Lister components (see Figure 9). The MenuBar class is the pull-down menu part of the panel. It is identified by the *File Edit Display* heading. The Face class covers the anchor name, type and orientation. The Control class is the eight buttons on the right hand side of the panel. The Lister class consists of the listing widget and *Jump* button. Figure 9 has been enlarged to show more detail.

The declaration of the Panel class is shown in Figure 10. As can be seen above, the Panel class is merely a container for the other four classes. Note the only operation the panel can perform is to update itself. This is done through a call to `updateYourself()`. Because the panel contains the other four classes, the panel must make sure that all four parts are updated as well. This is done through calls to the `updateYourself()` member

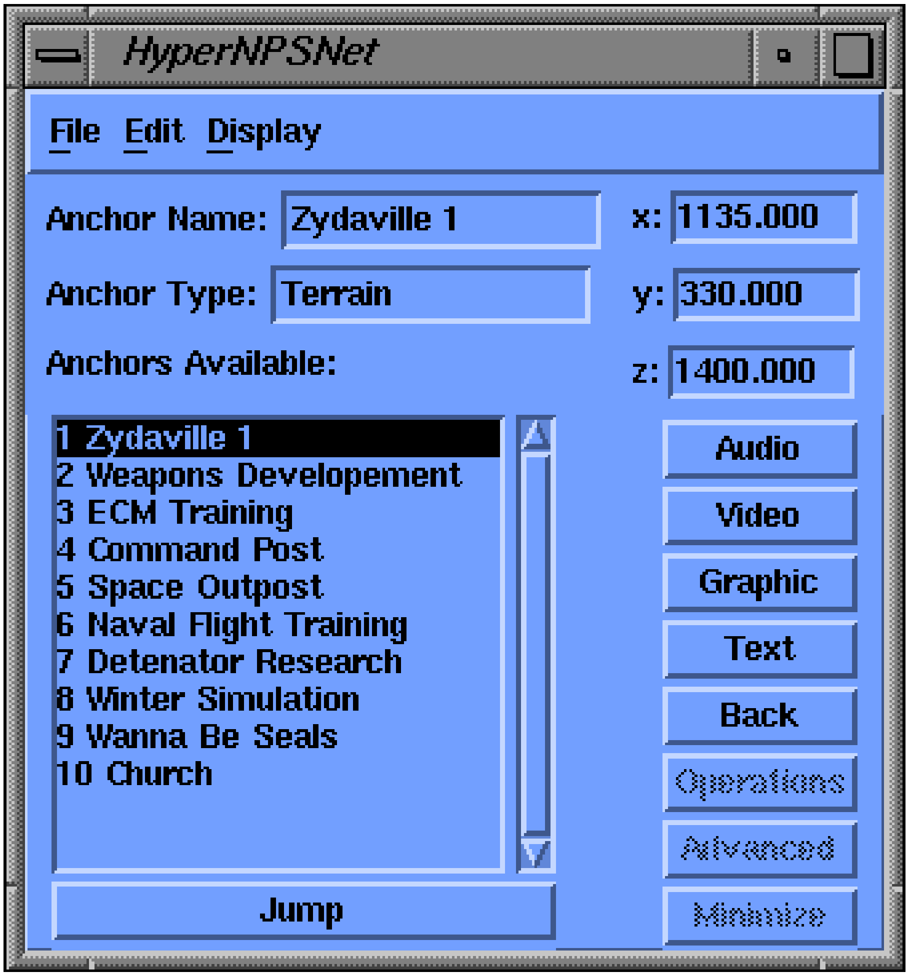

Figure 9: HyperNPSNET Control Panel

functions of the four contained classes. This is shown in the definition of the `updateYourself()` member function for the panel class (see Figure 11).

Various display properties of the panel are set using `_defaults[]`, which is used during Panel construction (see Figure 12). This is where the text strings are defined that appear on the Panel and other pop-up dialog boxes.

a. Menubar

In Figure 9, the menubar is identified by the thee pull down menus *File*, *Edit* and *Display*. All three pull down menus are shown in Figure 13. The three menus are

```
class Panel : public UIComponent {

private:

  MenuBar *_menuBar;                  // the menu bar
  Face *_face;                        // the anchor display portion
  Control *_control;                  // control panel
  Lister *_lister;                    // the anchor list

protected:

  static String _defaults[];

public:

  Panel ( Widget, char * );
  ~Panel();
  void updateYourself();
};
```

Figure 10: Panel Class Declaration

```
void Panel::updateYourself()
{
  // the panel merely updates all of its parts
  _face->updateYourself();
  _menuBar->updateYourself();
  _control->updateYourself();
  _lister->updateYourself();
}
```

Figure 11: `updateYourself()` member function for the Panel Class

constructed in the MenuBar constructor. For details on the actions of the MenuBar entities see the appendix. The MenuBar class declaration is shown in Figure 14. As with the other Panel components, the MenuBar is responsible for updating itself. This means modifying the displayed menus when the state of the simulator dictates it. These updates are applied to the *Display* pull down menu. For instance, when the terrain is textured, the menu displays “Turn Texturing Off”. If the user selects to turn off texturing by selecting the appropriate menu item, then the `displayTexturing()` member function is called (see Figure 15). This

```
String Panel::_defaults[] =
{
  “*borderWidth:                     0”,
  “*highlightThickness:              0”,
  “*face*cursorPositionVisible:      False”,
  “*control.audio.labelString:       Audio”,
  “*control.video.labelString:       Video”,
  “*control.graphic.labelString:     Graphic”,
  “*control.text.labelString:        Text”,
  “*control.back.labelString:        Back”,
  “*control.ops.labelString:         Operations”,
  “*control.advanced.labelString:    Advanced”,
  “*control.minimize.labelString:    Minimize”,
  “*list.jump.labelString:           Jump”,
  “*editAnchor.title:                HyperNPSNet Anchor Editor”,
  “*preferences.title:               HyperNPSNet User Preferences”,
  NULL,
};
```

Figure 12: `_defaults` for the Panel Class

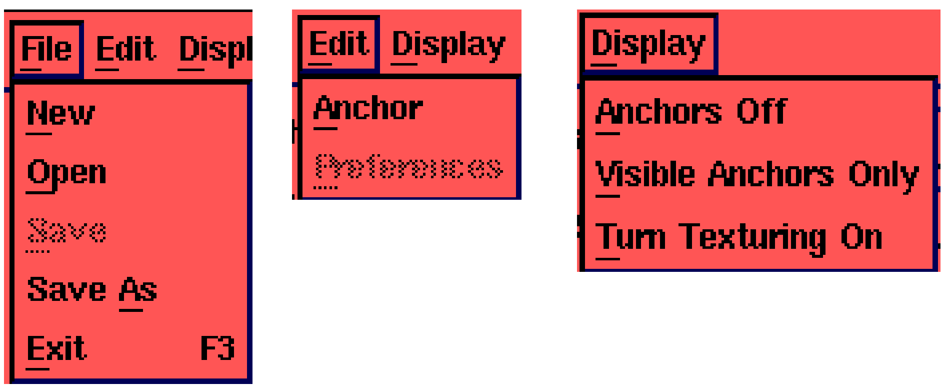

Figure 13: Hyper-NPSNET Control Panel Pull-down Menus

function first toggles the state of texturing and then calls `updateMenuBarState()`, a slimmed down version of `which` is shown in Figure 16. `updateMenuBarState()` insures the state of the pull-down menus are always up to date. This is a separate process from `updateYourself()` (see Figure 17), which directs the anchor editor to update itself (see “Anchor Editor Panel” in Section IV.B.2.).

```
class MenuBar : public BasicComponent {
private:
 Widget anchors;
 Widget localAnchors;
 Widget texture;
 Widget openButton;
 Widget saveButton;
 Widget saveAsButton;
 EditAnchor *_edit;
 Preferences *_prefs;
 unsigned editor_active;
 static void anchorCallback ( Widget, XtPointer, XtPointer );
 static void displayAnchorsCallback ( Widget, XtPointer, XtPointer );
 static void displayResetViewCallback (Widget, XtPointer, XtPointer);
 static void displayLocalAnchorsCallback ( Widget, XtPointer,
                                  XtPointer);
 static void displayTexturingCallback (Widget, XtPointer, XtPointer);
 static void exitCallback ( Widget, XtPointer, XtPointer );
 static void newCallback ( Widget, XtPointer, XtPointer );
 static void openCallback ( Widget, XtPointer, XtPointer );
 static void preferencesCallback ( Widget, XtPointer, XtPointer );
 static void promptOpenCallback ( Widget, XtPointer, XtPointer );
 static void promptSaveCallback ( Widget, XtPointer, XtPointer );
 static void saveCallback ( Widget, XtPointer, XtPointer );
 static void saveAsCallback ( Widget, XtPointer, XtPointer );
 void anchor();
 void displayAnchors();
 void displayResetView();
 void displayLocalAnchors();
 void displayTexturing();
 void myExit();
 void myNew();
 void myOpen();
 void preferences();
 void promptOpen ( XmSelectionBoxCallbackStruct * );
 void promptSave ( XmSelectionBoxCallbackStruct * );
 void save();
 void saveAs();
 void updateMenuBarState();
 void createFileMenu ( Widget, char* );
 void createEditMenu ( Widget, char* );
 void createDisplayMenu ( Widget, char* );
public:
 MenuBar ( Widget, char* );
 unsigned getEditorActive() { return editor_active; }
 void setEditorActive( Boolean state ) { editor_active = state; }
 void updateYourself();
};
```

Figure 14: MenuBar Class Declaration

```
void MenuBar::displayTexturing()
{
 if ( global.getDisplayTexturing() )
   global.setDisplayTexturing ( False );
 else
   global.setDisplayTexturing ( True );

 // now update the menubar state
 updateMenuBarState();
}
```

Figure 15: `displayTexturing()` for MenuBar Class

```
void MenuBar::updateMenuBarState()
{
 // removed all except for texturing stuff

 // set the label text for texturing on or off
 if ( global.getDisplayTexturing() )
   xmstr = XmStringCreateSimple ( “Turn Texturing Off” );
 else
   xmstr = XmStringCreateSimple ( “Turn Texturing On” );

 n = 0;
 XtSetArg ( args[n], XmNlabelString, xmstr ); n++;
 XtSetArg ( args[n], XmNmnemonic, ’T’); n++;
 XtSetValues ( texture, args, n);
}
```

Figure 16: Reduced version of `updateMenuBarState()` for MenuBar Class

b. Panel Face

Just below the MenuBar is the Face (see Figure 9). The face is responsible for displaying the current anchor name, its type and its coordinates. In Figure 9, the current anchor is the “Zydaville 1” anchor. It is located at coordinates (1135.0, 330.0, 1400.0) and has a TERRAIN type. When the current anchor changes, the face updates the displayed information to match. The Face class declaration shows the only public access functions

```
void MenuBar::updateYourself()
{
  // the only thing to do is make sure the anchor editor is updated
  _edit->updateYourself();

  // note the menubar status is constantly being updated anyway
  // and shouldn’t need to re-updated here
}
```

Figure 17: `updateYourself()` for MenuBar Class

are the constructor and the `updateYourself()` member function (see Figure 18). The `updateYourself()` member function is shown in Figure 19. Notice the line:

```
class Face: public BasicComponent {

private:

  Widget _label;                  // the “Anchors Available:” label
  Widget _namedata;               // the current anchor name
  Widget _namelabel;              // “Anchor Name:”
  Widget _typedata;               // the current anchor type
  Widget _typelabel;              // “Anchor Type:”
  Widget _xdata;                  // the x coord
  Widget _ydata;                  // the y coord
  Widget _zdata;                  // the z coord

  void setCoordX (char*);         // change the x coord
  void setCoordY (char*);         // change the y coord
  void setCoordZ (char*);         // change the z coord
  void setNameData (char*);       // change the name
  void setTypeData (char*);       // change the type

public:

  Face (Widget, char *);
  void updateYourself();
};
```

Figure 18: Face Class Declaration

```
void Face::updateYourself()
{
 // get the current ID
 unsigned id = global.getCurrentAnchorID();

 // if id == 0 blank everything out
 if ( id == 0 ) {
   setNameData ( ““ );
   setTypeData ( ““ );
   XmTextFieldSetString ( _xdata, ““ );
   XmTextFieldSetString ( _ydata, ““ );
   XmTextFieldSetString ( _zdata, ““ );
 }

 else {
   // get the anchor info that relates to the face
   AnchorPtr a = global.getHypersystem().GetAnchorPtr ( id );
   char *name = a->getName();
   unsigned type = a->getType();
   float *coords = a->getCoords();

   // set the name
   setNameData ( name );

   // set the type
   switch ( type ) {
   case ANCHOR_TERRAIN:
     setTypeData ( ( char * ) ANCHOR_TERRAIN_STRING );
     break;
   default:
     setTypeData ( ( char * ) ANCHOR_UNKNOWN_STRING );
     break;
   }

   // set the coords
   char buf[10];
   sprintf ( buf, “%6.3f”, coords[0] );
   setCoordX ( buf );
   sprintf ( buf, “%6.3f”, coords[1] );
   setCoordY ( buf );
   sprintf ( buf, “%6.3f”, coords[2] );
   setCoordZ ( buf );
 }
}
```

Figure 19: `updateYourself()` for Face Class

```
AnchorPtr a = global.getHypersystem().GetAnchorPtr ( id );
```

Here, an instance of the global class is used to get a pointer to the Hyper-System. This pointer is then used to get a pointer to the current anchor. The anchor pointer is used in subsequent lines in Figure 19 to retrieve the anchor name, type and coordinates. This style is used throughout the application whenever the GUI side of the program needs any Hyper-System information.

c. Control Class

The Control class consists of the eight buttons on the right hand side of the Hyper-NPSNET panel (see Figure 9). The top four buttons operate on the audio, video, graphic and text information. That is, when any of the buttons are pressed, the appropriate action is taken. Depending on the button, this may mean to play an audio track, or view a video clip, or to view a still image or text file. As with the other classes, the Control class must monitor the current anchor id and make the appropriate buttons sensitive or insensitive based on whether that particular type of information is currently available. The class declaration is shown in Figure 20, followed by `updateYourself()` in Figure 21.

The basic test for all the media buttons consists of checking if a valid filename has been attached to the node. Currently this means any non-NULL filename. The existence of the file is not checked as this should be done at the time the filename is first attached to the node. As for the operation of the BACK button, as long as there is a current anchor id available, then the button is made sensitive. If there is no current anchor id, then all of the control buttons are insensitized.

d. Lister Class

The Lister class is the final piece of the main panel. The Lister is responsible for displaying a list of available anchors. The anchors are displayed by their name and are arranged by anchor id in an increasing down fashion. Any list item is selectable with the mouse by either clicking once followed by clicking the *Jump* button, or by double clicking on the list item. Upon setting a new current anchor, the entire main panel will update itself

```
class Control : public BasicComponent {
private:
  Widget _advancedWidget;              // advanced button
  Widget _audioWidget;                 // audio button
  Widget _backWidget;                  // back button
  Graphic *_graphic;                   // used to view images
  Widget _graphicWidget;               // graphic button
  Widget _minimizeWidget;              // minimize button
  Widget _opsWidget;                   // operations button
  TextBox *_text;                      // a popup scrolled text guy
  Widget _textWidget;                  // text button
  Widget _videoWidget;                 // video button
  void createControlButtons();
  void registerControlButtonCallbacks();
  void audio();                        // Called onaudio button hit
  void video();                        //          video
  void graphic();                      //          graphic
  void text();                         //          text
  void back();                         //          back
  void ops();                          //          operations
  void advanced();                     //          advanced
  void minimize();                     //          minimize

  // Static member functions that interface the above member functions
  // with Motif widget callbacks
  static void audioCallback ( Widget, XtPointer, XtPointer );
  static void videoCallback ( Widget, XtPointer, XtPointer );
  static void graphicCallback ( Widget, XtPointer, XtPointer );
  static void textCallback ( Widget, XtPointer, XtPointer );
  static void backCallback ( Widget, XtPointer, XtPointer );
  static void opsCallback ( Widget, XtPointer, XtPointer );
  static void advancedCallback ( Widget, XtPointer, XtPointer );
  static void minimizeCallback ( Widget, XtPointer, XtPointer );
  char *Control::getCurrentAnchorNodeFilename ( unsigned type );
  void insensitizeEverything();
  void setAudioInsensitive() { XtSetSensitive ( _audioWidget, False ); }
  void setAudioSensitive() { XtSetSensitive ( _audioWidget, True ); }
  void setBackInsensitive() { XtSetSensitive ( _backWidget, False ); }
  void setBackSensitive() { XtSetSensitive ( _backWidget, True ); }
  void setGraphicInsensitive()
                    { XtSetSensitive ( _graphicWidget, False ); }
  void setGraphicSensitive() { XtSetSensitive ( _graphicWidget, True ); }
  void setTextInsensitive() { XtSetSensitive ( _textWidget, False ); }
  void setTextSensitive() { XtSetSensitive ( _textWidget, True ); }
  void setVideoInsensitive() { XtSetSensitive ( _videoWidget, False ); }
  void setVideoSensitive() { XtSetSensitive ( _videoWidget, True ); }

public:
  Control ( Widget, char * , Panel * );
  void updateYourself();
};
```

Figure 20: Control Class Declaration

```
void Control::updateYourself()
{
  // get the current ID
  unsigned id = global.getCurrentAnchorID();

  // if id == 0 insensitize all the buttons
  if ( id == 0 ) {
    setAudioInsensitive();
    setVideoInsensitive();
    setGraphicInsensitive();
    setTextInsensitive();
    setBackInsensitive();
  }

  else {
    // turn on only the buttons that need to be on
    // what this means is check all nodes for the current anchor to see
    // if they have a filename associated with them. If no filename, then
    // leave the button insensitive

    if ( strcmp ( getCurrentAnchorNodeFilename ( HYPER_AUDIO ), "" ) == 0 )
      setAudioInsensitive();
    else
      setAudioSensitive();

    if ( strcmp ( getCurrentAnchorNodeFilename ( HYPER_GRAPHIC ), "" )
                                            == 0 )
      setGraphicInsensitive();
    else
      setGraphicSensitive();

    if ( strcmp ( getCurrentAnchorNodeFilename ( HYPER_VIDEO ), "" ) == 0 )
      setVideoInsensitive();
    else
      setVideoSensitive();

    if ( strcmp ( getCurrentAnchorNodeFilename ( HYPER_TEXT ), "" ) == 0 )
      setTextInsensitive();
    else
      setTextSensitive();

    // if the current id != 0 then activate the back button
    setBackSensitive();
  }

  // make sure to update any of the objects that need updating
  _text->updateYourself();
}
```

Figure 21: `updateYourself()` for Control Class

as previously described. The Lister must always insure that the current anchor is highlighted whenever the user has not clicked on a non-current anchor in the list. In addition, the *Jump* button must always be made sensitive whenever an anchor is being selected. The class declaration is shown in Figure 22, and `updateYourself()` is shown in Figure 23.

```
class Lister : public BasicComponent {

private:

  Widget _lister;                    // the scrolled list
  Widget _scroll;                    // surrounds _list
  Widget _jumpWidget;                // the jump button

  unsigned _tempAnchor;              // holds temp anchor id from
                                     // single selection

  void jump();                        // called on "jump"
  void doubleClick ( XmListCallbackStruct *);
  void setJumpInsensitive() { XtSetSensitive ( _jumpWidget,
                                     False); }
  void singleSelection ( XmListCallbackStruct *);

  static void doubleClickCallback ( Widget, XtPointer, XtPointer );
  static void jumpCallback ( Widget, XtPointer, XtPointer );
  static void singleSelectionCallback ( Widget, XtPointer,
                                     XtPointer );

  void deleteAllListItems() { XmListDeleteAllItems ( _lister ); }
  void setSelection ( char* );

public:

  Widget getListWidget() { return _lister; }
  Lister ( Widget, char * );
  void updateYourself();
};
```

Figure 22: Lister Class Declaration

2. Anchor Editor Panel

The anchor editor panel is used to display more detailed information about the current anchor and for authoring or editing purposes. The panel layout shows three distinct

```
void Lister::updateYourself()
{
  // update the entire list
  // start at the beginning of the systems anchorlist
  deleteAllListItems();

  // for each anchor in the list ...
  for(register i = 0; i < global.getHypersystem().NAnchors(); i++) {
    // get an anchor
    c_anchor = global.getHypersystem().AnchorList().current_data();
    id = c_anchor->getId();
    name = c_anchor->getName();

    // make the string
    sprintf(string, "%u %s", id, name);
    xmstr = XmStringCreateSimple ( string );

    // add this anchors string to the listview
    XmListAddItem( global.getPanel()->getListWidget(), xmstr, 0 );

    // free the XmString
    XmStringFree ( xmstr );

    // move on the the next anchor
    global.getHypersystem().AnchorList().next();
  }

  // get the current ID and blank selection if == 0
  id = global.getCurrentAnchorID();
  if ( id == 0 )
  XmListDeselectAllItems ( _lister );

  else {
    // get the anchor from the id
    AnchorPtr a = global.getHypersystem().GetAnchorPtr ( id );

    // get the name
    name = a->getName();

    // make the selection string and select it
    char string[40];
    sprintf(string, "%u %s", id, name);
    setSelection ( string );
 }
}
```

Figure 23: `updateYourself()` for Lister Class

areas (see Figure 24). The top area displays the anchor name, type, orientation and coordinates. The middle area lists the filenames associated with the audio, video, graphics and textual information attached to the anchor. The bottom part is the button bar where changes are saved or ignored. The panel can also be closed using the *Close* button. New anchors can be created and added with the *New Anchor* and *Add Anchor* buttons.

Unlike the main panel, the anchor editor panel consists of only one C++ class. The class declaration is shown in Figure 25. In the class declaration, one can see the

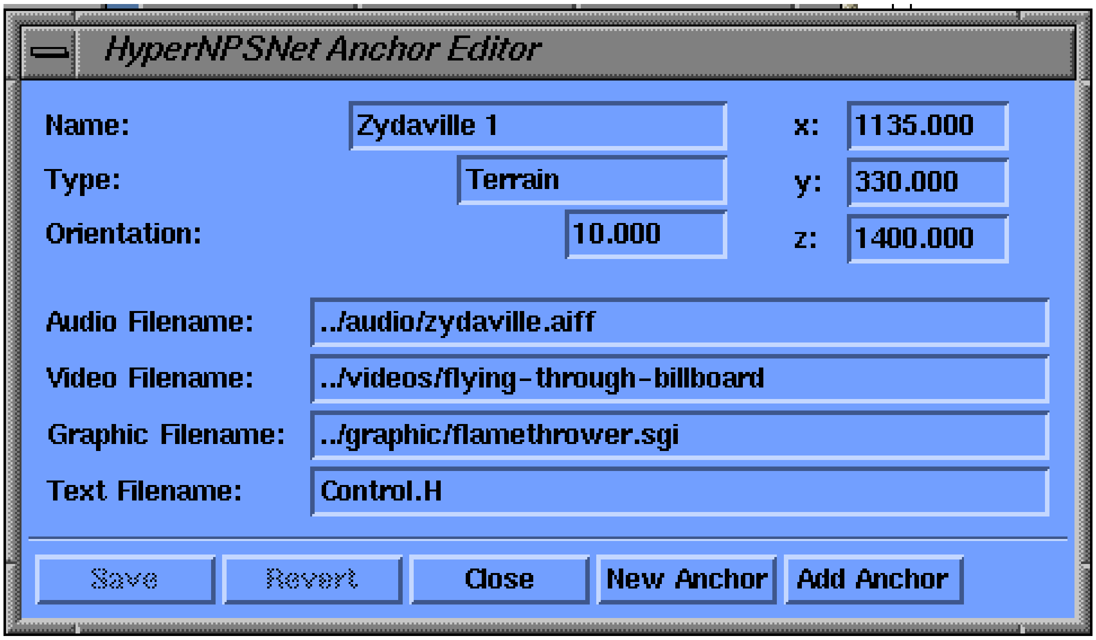

Figure 24: The Anchor Editor Panel

individual widget parts that make up the panel. For instance, the individual widgets that make up the button bar are: `_addAnchor`, `_close`, `_newAnchor`, `_revert` and `_save`. There are a number of private member functions but the only public ones are for popping the panel up and down and for updating the display. As with the components of the main panel, the anchor editor is responsible for self maintenance, that is, it must always display the correct information. This is done through a call to `updateYourself()`(see Figure 26). There are three conditions checked for by `updateYourself()`. The first case is where no anchor has been selected. In this case, the editor panel will display all blank fields. The second is for the creation of a new anchor. In this case, the fields are filled with “New anchor” information awaiting the user to overwrite them with the correct information. The

```
class EditAnchor : public BasicComponent {
private:
  Widget _addAnchor;                   // duplicate current anchor as new
  Widget _audio;                       // the audio filename
  Widget _close;                       // the close/cancel button
  Widget _graphic;                     // the graphic filename
  Widget _nameData;                    // the name textfield
  Widget _newAnchor;                   // button to create new anchor
  unsigned _newAnchorState;            // is a new anchor being created?
  Widget _orientationData;             // the orientation textfield
  Widget _revert;                      // the undo all button
  Widget _save;                        // the save button
  unsigned _saveState;                 // is saving necessary or not?
  Widget _text;                        // the text filename
  Widget _typeData;                    // the type textfield
  Widget _video;                       // the video filename
  Widget _xData;                       // the x coordinate
  Widget _yData;                       // the y coordinate
  Widget _zData;                       // the z coordinate

  void addAnchor();
  static void addAnchorCallback ( Widget, XtPointer, XtPointer );
  void close();
  static void closeCallback ( Widget, XtPointer, XtPointer );
  void newAnchor();
  static void newAnchorCallback ( Widget, XtPointer, XtPointer );
  void revert();
  static void revertCallback ( Widget, XtPointer, XtPointer );
  void save();
  static void saveCallback ( Widget, XtPointer, XtPointer );
  void setAudio ( char* );
  void setCoordX ( char* );
  void setCoordY ( char* );
  void setCoordZ ( char* );
  void setGraphic ( char* );
  void setName ( char* );
  void setOrientation ( char* );
  void setText ( char * );
  void setType ( char* );
  void setValueChangeCallback();
  void setVideo ( char* );
  void updateEditorState();
  void valueChanged();
  static void valueChangedCallback ( Widget, XtPointer, XtPointer );

public:
  EditAnchor ( Widget, char* );
  void showYourself();
  void updateYourself();
};
```

Figure 25: EditAnchor Class Declaration

```
void EditAnchor::updateYourself()
{
  // get the current ID
  nsigned id = global.getCurrentAnchorID();

  // if id == 0 blank everything out
  if ( id == 0 && _newAnchorState == EDIT_NEW_ANCHOR_NOT_PENDING ) {
    setName ( "" );
    setType ( "" );
    setOrientation ( "" );
    setCoords ( "" );
    setAudio ( "" );
    setVideo ( "" );
    setGraphic ( "" );
    setText ( "" );
  }
  else if ( _newAnchorState == EDIT_NEW_ANCHOR_PENDING ) {
    setName ( "New Anchor" );
    setType ( ( char * ) ANCHOR_TERRAIN_STRING );
    setOrientation ( "0.0" );
    setCoords ( "0.0" );
    setAudio ( "New Audio" );
    setVideo ( "New Video" );
    setGraphic ( "New Graphic" );
    setText ( "New Text" );
  }
  else {
    // get the anchor
    AnchorPtr a = global.getHypersystem()->GetAnchorPtr ( id );
    char *name = a->getName();
    unsigned type = a->getType();
    float *coords = a->getCoords();
    setName ( name );
    switch ( type ) {
    case ANCHOR_TERRAIN:
      setType ( ( char * ) ANCHOR_TERRAIN_STRING );
      break;
    default:
      setType ( ( char * ) ANCHOR_UNKNOWN_STRING );
      break;
    }

    // set the orientation
    float orientation = a->getOrientation();
    char buf[10];
    sprintf ( buf, "%6.3f", orientation );
    setOrientation ( buf );
```

Figure 26: `updateYourself()` for the EditAnchor Class

```
    // set the coords
    sprintf ( buf, "%6.3f", coords[0] );
    setCoordX ( buf );
    sprintf ( buf, "%6.3f", coords[1] );
    setCoordY ( buf );
    sprintf ( buf, "%6.3f", coords[2] );
    setCoordZ ( buf );

    // set the node filenames
    // audio first
    unsigned nodeid = a->getAudio();
    HNode node = (global.getHypersystem())->getHNode ( nodeid );
    setAudio ( node->getFilename() );

    // now video
    nodeid = a->getVideo();
    node = global.getHypersystem()->getHNode ( nodeid );
    setVideo ( node->getFilename() );

    // now graphic
    nodeid = a->getGraphics();
    node = global.getHypersystem()->getHNode ( nodeid );
    setGraphic ( node->getFilename() );

    // and last, text
    nodeid = a->getText();
    node = global.getHypersystem()->getHNode ( nodeid );
    setText ( node->getFilename() );
  }

  // set the save state to EDIT_SAVE_NOT_NEEDED
  _saveState = EDIT_SAVE_NOT_NEEDED;
  updateEditorState();
}
```

Figure 26:`updateYourself()` for the EditAnchor Class (Continued)

third case is the general case, where a current anchor id exists and so `updateYourself()` retrieves the correct information from the hypersystem and fills the fields accordingly.

3. User Preferences Panel

The user preferences panel allows the user to specify the operating characteristics of certain features of the program (see Figure 27). This panel consists of four distinct regions separated by horizontal and vertical bars. The upper left region is the *Anchor Auto View* region. In this region the user specifies any media formats that are to be displayed automatically whenever the user's eye position is within the distance specified by the

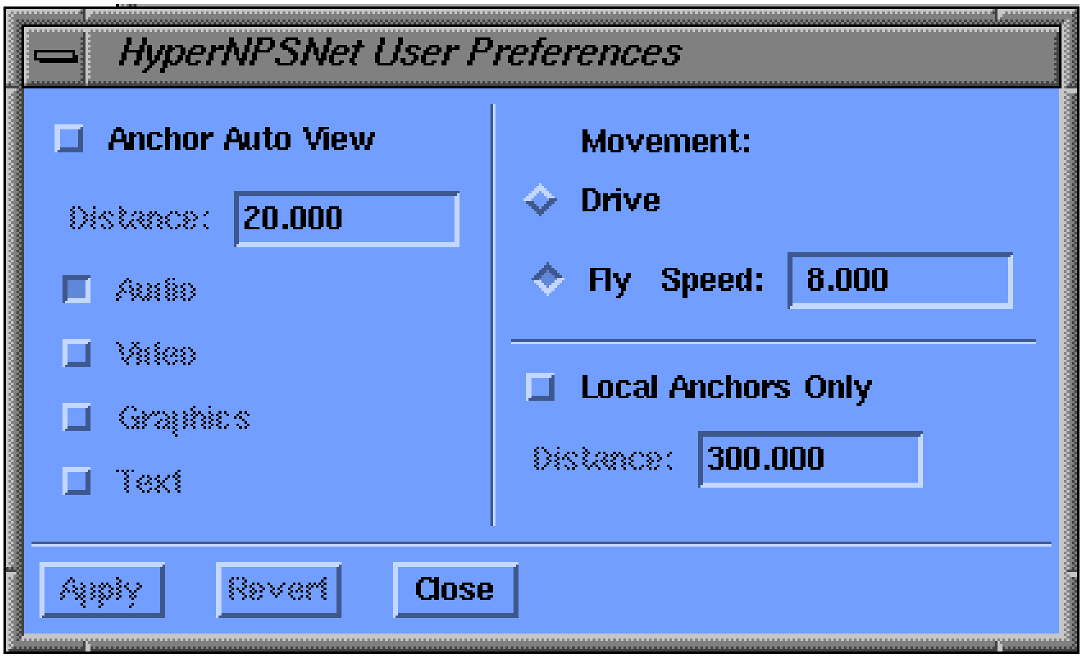

Figure 27: User Preferences Panel

Distance field to an anchor. In Figure 27 the distance specified is 20 meters. Notice that the *Anchor Auto View* button will turn on the sensitivity of the *Audio*, *Video*, *Graphics* and *Text* buttons. The upper right region is the movement type region. Here the user specifies whether the vehicle being operated is a ground vehicle or a flying vehicle[1]. If flying, the speed can be modified through the *Speed* text field widget. Right below the movement area is the *Local Anchors Only* area. Here the user specifies whether all information anchors are displayed or only ones close to the users eye position. The distance that specifies what is local can be changed through the *Distance* text field widget. The last region in the panel is the button bar. This is where the user applies the changes made or ignores them and lets the preferences revert back to the previously saved state. The panel can also be closed from the button bar.

As with the Anchor Editor panel, the User Preferences panel is a single C++ class. The class declaration is shown in Figure 28. There are a number of individual widgets

1. Currently only flying vehicles are implemented.

```
class Preferences : public BasicComponent {
private:
  Widget _apply;                       // the apply button
  Widget _audio;                       // the audio radio button
  Widget _autoViewToggle;              // the anchor auto view button
  Widget _autoDistanceData;            // the auto distance text field
  Widget _autoDistanceLabel;           // the auto distance label
  Widget _close;                       // the close/cancel button
  Widget _driveToggle;                 // the drive toggle button
  Widget _flyToggle;                   // the fly toggle button
  Widget _graphics;                    // the graphic radio button
  Widget _localDistanceData;           // the local distance text field
  Widget _localDistanceLabel;          // local distance label and data RC
  Widget _localToggle;                 // local anchors only toggle button
  Widget _revert;                      // the undo all button
  unsigned _saveState;                 // is saving necessary or not
  Widget _speedLabel;                  // the speed label guy
  Widget _speedData;                   // the speed text field
  Widget _text;                        // the text radio button
  Widget _video;                       // the video radio button

  void apply();
  static void applyCallback ( Widget, XtPointer, XtPointer );
  void autoView();
  static void autoViewCallback ( Widget, XtPointer, XtPointer );
  void close();
  static void closeCallback ( Widget, XtPointer, XtPointer );
  void createButtons ( Widget, char* );
  void  createTopInfo ( Widget, char* );
  void  drive();
  static void driveCallback ( Widget, XtPointer, XtPointer );
  void  fly();
  static void flyCallback ( Widget, XtPointer, XtPointer );
  void local();
  static void localCallback ( Widget, XtPointer, XtPointer );
  void  registerCallbacks();
  void  removeCallbacks();
  void revert();
  static void revertCallback ( Widget, XtPointer, XtPointer );
  void  updatePreferencesState();
  void valueChanged();
  static void valueChangedCallback ( Widget, XtPointer, XtPointer );

public:
  Preferences ( Widget, char* );
  void showYourself();
  void updateYourself();
};
```

Figure 28: Preferences Class Declaration

that make up the panel and this is reflected in the private widget variables. As in most of the other classes, there is limited public access to this class, namely `showYourself()` and `updateYourself()`. `showYourself()` is shown in Figure 29. `showYourself()` is

```
void Preferences::showYourself()
{
  // toggle the visibility of the preferences panel
  if ( XtIsRealized ( _w ) ) {

    XtPopdown ( _w );
    XtUnrealizeWidget ( _w );
  }

  else {
    XtRealizeWidget ( _w );
    XtPopup ( _w, XtGrabNone );

    // set the state variables
    _saveState = PREF_APPLY_NOT_NEEDED;

    // remove all the callbacks while your opening the tool
    removeCallbacks();

    // now update all fields of the panel
    updateYourself();

    // now register the callbacks
    registerCallbacks();
  }
}
```

Figure 29: `showYourself()` for Preferences Class

responsible for popping the panel up and down when requested. `updateYourself()` is shown in Figure 30. `updateYourself()` ensures that the displayed user preferences match values stored within the Global class slots and that the state of the buttons matches the current operation.

```
void Preferences::updateYourself()
{
  // set the auto view distance
  float value = global.getAutoViewRange();
  char buf[20];
  sprintf ( buf, “%6.3f”, value );
  XmTextFieldSetString ( _autoDistanceData, buf );
  XmTextFieldSetInsertionPosition ( _autoDistanceData, 0 );

  // set audio, video, graphics and text toggle buttons as appropriate
  if ( global.autoAudio() )
    XmToggleButtonSetState ( _audio, True, False );
  else
    XmToggleButtonSetState ( _audio, False, False );
  if ( global.autoVideo() )
    XmToggleButtonSetState ( _video, True, False );
  else
    XmToggleButtonSetState ( _video, False, False );
  if ( global.autoGraphics() )
    XmToggleButtonSetState ( _graphics, True, False );
  else
    XmToggleButtonSetState ( _graphics, False, False );
  if ( global.autoText() )
    XmToggleButtonSetState ( _text, True, False );
  else
    XmToggleButtonSetState ( _text, False, False );

  // set the flying speed
  value = global.getFlyingSpeed();
  sprintf ( buf, “%6.3f”, value );
  XmTextFieldSetString ( _speedData, buf );
  XmTextFieldSetInsertionPosition ( _speedData, 0 );

  // set the local anchor distance
  value = global.getLocalAnchorsRange();
  sprintf ( buf, “%6.3f”, value );
  XmTextFieldSetString ( _localDistanceData, buf );
  XmTextFieldSetInsertionPosition ( _localDistanceData, 0 );

  // check the auto view mode
  if ( global.getAutoViewMode() == True ) {

    // set the anchor auto view button on
    XmToggleButtonSetState ( _autoViewToggle, True, False );

    // make sure all the auto view guys are sensitive
    XtSetSensitive ( _autoDistanceData, True );
    XtSetSensitive ( _autoDistanceLabel, True );
    XtSetSensitive ( _audio, True );
    XtSetSensitive ( _video, True );
    XtSetSensitive ( _graphics, True );
```

Figure 30: `updateYourself()` for the Preferences Class

```
    XtSetSensitive ( _text, True );
  }
  else {
    // set anchor auto view toggle button off
    XmToggleButtonSetState ( _autoViewToggle, False, False );
    // insensitize the rest of the auto view stuff
    XtSetSensitive ( _autoDistanceData, False );
    XtSetSensitive ( _autoDistanceLabel, False );
    XtSetSensitive ( _audio, False );
    XtSetSensitive ( _video, False );
    XtSetSensitive ( _graphics, False );
    XtSetSensitive ( _text, False);
  }
  // check the movement type
  if ( global.getLocomotionMode() == GLOBAL_DRIVE_MODE ) {
    // set drive toggle button on
    XmToggleButtonSetState ( _driveToggle, True, False );
    // set fly toggle button off
    XmToggleButtonSetState ( _flyToggle, False, False );
    // insensitize the speed guy
    XtSetSensitive ( _speedLabel, False );
    XtSetSensitive ( _speedData, False );
  }
  else {
    // set drive toggle button off
    XmToggleButtonSetState ( _driveToggle, False, False );
    // set fly toggle button on
    XmToggleButtonSetState ( _flyToggle, True, False );
    // sensitize the speed guy
    XtSetSensitive ( _speedLabel, True );
    XtSetSensitive ( _speedData, True );
  }

  // check the local anchors guy
  if ( global.getLocalAnchors() == TRUE ) {
    // set the local anchors toggle on
    XmToggleButtonSetState ( _localToggle, True, False );
    // sensitize the distance rc guy
    XtSetSensitive ( _localDistanceData, True );
    XtSetSensitive ( _localDistanceLabel, True );
  }
  else {
    // set the local anchors toggle off
    XmToggleButtonSetState ( _localToggle, False, False );
    // insensitize the distance rc guy
    XtSetSensitive ( _localDistanceData, False );
    XtSetSensitive ( _localDistanceLabel, False );
  }

  // now make sure the state of the buttons is correct
  updatePreferencesState();
}
```

Figure 30: `updateYourself()` for the Preferences Class (Continued)

V. HYPERMEDIA AUTHORING

Unfortunately, it is difficult to design typical hypertext documents due to the difference in document structure from classical linear text [Nielsen90]. This statement is also true for hypermedia documents, but even goes a step further. To design hypermedia documents that are informative, the media used must be relevant. This is easier said than done. When writing a text based document, the author comes up with the words, but when writing a hypermedia document, you may need audio clips, video segments and graphics in addition to the text. Even though there are quite a number of available video and audio samples available, what one really needs is the capability to grab any audio or video segments through a variety of means and make them hypermedia capable. By this I mean get them in some digital format and store them as a file in a computer. The capability to do this is just now becoming common, but until hypermedia authoring systems are also common place, creating useful hypermedia documents will be a difficult task.

A. AUTHORING WITH HYPER-NPSNET

The author system in Hyper-NPSNET is limited but easy to use. In the case of Hyper-NPSNET, authoring means the ability to designate where information anchors are placed, what the orientation of the anchor is and what multimedia files will be associated with that anchor. All of this is accomplished through the Anchor Editor (see "Anchor Editor Panel" in Section IV.B.2.). The Anchor Editor is used not only for displaying and changing anchor attributes, it is also used for the creation of new anchors. New anchors can be added to an existing *world* database or a new *world* can be created.

1. Creating A New World

To create a new world, the program is started as usual (see "STARTING AND EXITING THE PROGRAM" in the Appendix). Instead of loading an existing world database, bring up the Anchor Editor panel by selecting "Anchor" from the "Edit" pull down menu on the main Hyper-NPSNET control panel. Since no database has been entered and there is no current anchor, all fields in the Anchor Editor will be blank.

To initiate the new anchor, select the "New Anchor" button at the bottom of the panel. Once pressed, all fields on the editor will be filled in with default information. This default information currently sets the anchor location to (0.0, 0.0, 0.0) with an orientation of 0.0 degrees. The default name given is "New Anchor" and the anchor type is set to "Terrain". The default multimedia file names are set to "New Audio", "New Video", "New Graphic" and "New Text" for the audio, video, graphics and text nodes respectively.

At this point, using the mouse and keyboard, the author overrides these defaults with his own preferences. When all attributes of the new anchor are satisfactory, the author presses the "Save" button. This new anchor will be added to the world and will appear in the scrolling anchor list in the Hyper-NPSNET control panel. As this is done, all fields in the Anchor Editor will be set back to the default values awaiting input of the next anchor by the author. The editor continues to accept new anchors in this fashion until the author presses the "Revert" button. This operation will take the Anchor Editor out of the New Anchor mode and put it back into the edit current anchor information mode.

2. Adding New Anchors To An Existing World

To add a new anchor to a world database, start the program and load the world. After the world is loaded, bring up the Anchor Editor as indicated above. From this point on, the operations are the same as the section above. When the author selects the "New Anchor" button, all the same default information will be set. Upon filling in all the fields, the author saves the new anchor to the world by pressing the "Save" button. To add another anchor, update all the fields and save again. Continue this until all new anchors have been saved back to the world. Revert or Cancel to end the New Anchor adding.

3. Saving The World

If any new anchors are added to the world or if a new world is created, the world database must be saved to a file to allow the same world to be loaded again. To save the world, use the “File” pull down menu on the main Hyper-NPSNET control panel. Select “Save As” and a small dialog box will pop up requesting a file name to save the world to. This box must be used before any other operation can be performed. In fact the window manager will not allow any other operations to be performed until the world is saved or the “Save As” operation is cancelled.

The world is saved in an ascii format and is readable and editable by the author. Caution must be taken when modifying any world database with a text editor. It is best to make all changes through the use of the Anchor Editor, but certain types of changes can be easily and quickly made to the world through this ascii file. These include changes to any media file name, changes to any anchor’s coordinates or orientation and changes to the node id’s associated with an anchor. New anchors or nodes should not be added to the world through this ascii file.

A sample world database file is shown in Figure 31. The actual file is 267 lines long, so only the first anchor and the associated 4 media nodes are shown. The association between the anchors and the media nodes is through temporary node id’s. As the file is read in by the system, all the links between an anchor and it’s nodes are automatically established.

```
BeginAnchorData
TotalAnchorCount 10

BeginAnchor 1
Name: Zydaville 1
Type: Terrain
Coords: 1135.000000 330.000000 1400.000000
Orientation: 10.000000
AudioNodeID: 1
VideoNodeID: 2
GraphicNodeID: 3
TextNodeID: 4
EndAnchor 1

...

BeginNodeData
TotalNodeCount 40

BeginNode 1
Type: Audio
Filename: /n/elsie/work3/lombardo/hyper/audio/zydaville.aiff
EndNode 1
BeginNode 2
Type: Video
Filename: /usr/zurich/videos/flying-through-billboard
EndNode 2
BeginNode 3
Type: Graphic
Filename: ../graphic/flamethrower.sgi
EndNode 3
BeginNode 4
Type: Text
Filename: Control.H
EndNode 4
...
```

Figure 31: Sample Hyper-NPSNET Ascii World Database File

VI. RESULTS

The main focus of this work is the design and implementation of underlying data structures to embed multimedia information in a real-time 3D virtual world. The hypersystem and GUI data structures are discussed in detail in Chapter IV. Discussion of results in this chapter revolves around three main areas. The first concerns the minimum capabilities of the software required to have a useful system. The second deals with creation and implementation of hypermedia databases that can be easily incorporated into training scenarios. The last deals with hardware performance for Hyper-NPSNET.

A. MINIMUM SOFTWARE CAPABILITY

The software capabilities of the system should include at a minimum: interactive navigation through the 3D virtual world, anchor selection within the 3D virtual world and consistency across the user interface. The navigation method chosen uses a typical 2D mouse. A first time user of the system, therefore, can easily pick up how to move around within the virtual world in a matter of seconds. This type of ease of use is necessary for a good user interface.

The operation of Hyper-NPSNET includes user selection of anchors and information nodes. Since the user is navigating through a 3D virtual world where 3D information anchors are viewable, anchors need to be selectable directly off the screen. This is accomplished, in Hyper-NPSNET, with the mouse using a “Point and Click” approach. This is a very intuitive and common way of interacting with computer applications.

A significant feature of any application is a consistent user interface. The pull-down menus and pop-up panels of Hyper-NPSNET are common components to user interfaces of window-based applications on a variety of platforms. This familiarity instills confidence in the user that he or she understands the flow of operations being performed. Even though general guidelines exist for the design of user interfaces [Mackinlay91], it was found that the pop-up panels tended to get somewhat more cluttered than desired. On top of this, there were no graphics objects at all within the panels. This would crowd the panels even more.

Not only is the layout of the interface panels important, the intelligence of the interface is equally significant. For instance, in the User Preferences panel (see "User Preferences Panel" in Section IV.B.3.), disabling the *Anchor Auto View* mode de-sensitizes the *Audio, Video, Graphics* and *Text* buttons, in addition to the *Distance* type in text field. This same idea was carried to the *Movement* and *Local Anchors Only* part of the same panel. All the panels in Hyper-NPSNET have smart buttons that can change their label and sensitivity dynamically. All writable text fields allow copy and paste type operations from both inside and outside of Hyper-NPSNET. All these features help make the user interface comfortable and therefore the application more usable.

B. HYPERMEDIA DATABASE CREATION

Relevant hypermedia "documents" or databases are difficult to generate. For instance, before a video sequence can be attached to an anchor, it must be produced in some format, typically VHS. The VHS video is then captured, either partially or completely, using some sort of dedicated hardware. This hardware could include an NTSC to RGB decoder or a video input equipped computer such as certain Silicon Graphics Indigo Elans. Once the video sequence is in a file format, it may undergo one or more rounds of editing before the content is deemed adequate. This procedure may need to be done on most if not all of the hypermedia database video sequences. This equates to a lot of time and effort. A similar argument holds for the Audio nodes as well as the Graphics nodes. This is still true for the Text nodes, but to a lessor extent.

C. HARDWARE PERFORMANCE

The performance of the underlying hardware is critical to the user "satisfaction" when using Hyper-NPSNET. A minimal hardware set necessarily includes audio and video capability. This is obvious from the nature of the program. Beyond this, the faster the machine the better.

The majority of the development and running of Hyper-NPSNET is done on Silicon Graphics Indigo Elan workstations. The Elans have full audio and video capability and for the most part their performance in this respect is satisfactory. The shortcomings are in general rendering speeds. The Elan’s proved to be too slow to run Hyper-NPSNET with a textured terrain. When texturing is turned off, all objects in the world including buildings, anchors, trees and the terrain are defined with materials with appropriate lighting. The Elan can handle this with frame rates in the range of about 10/sec. Once the terrain is textured, the frame rate drops to about 0.3/sec.

It is preferable to run the program with texturing turned on in order to create a more realistic virtual environment, but the frame rate is completely unsatisfactory. The solution to this problem is to run Hyper-NPSNET on a faster machine. With the program running on a Silicon Graphics Reality Engine with fully textured terrain, the frame rate is in the 20/sec range. Currently, the shortcomings of the Reality Engine is the lack of audio support. What is needed is full audio support at the high end machine level. This is being pursued by both Silicon Graphics and a third party hardware manufacturer.

When demos of Hyper-NPSNET are given, they are given first on the Reality Engine to show the high quality textured terrain and smooth motion, and then on the Indigo Elan to experience the audio aspect of the multimedia. When the high end machines support audio, the demos will not have this discontinuity and will be more impressive.

VII. FUTURE WORK

The main focus of this work was in proof of concept. This leaves a lot of room for improvement and future work. This chapter presents some of the near term changes needed to make Hyper-NPSNET more effective as a training and authoring tool.

A. DATABASE FRONT END

As the number of anchors in any world database increases, it gets difficult to administer the vast amounts of information available to the user. Currently the only interface to the hyper system, other than in the 3D world, is through the main control panel, the Anchor Editor panel or the User Preferences panel. This situation can be improved with the addition of a sophisticated user interface to the hyper system database.

The database front end would allow the user to make queries into the hyper system. For instance, if the user wanted to know all anchors that had video clips relevant to current enemy tank positions, he could *ask* the system and be given a mouse sensitive list that he could use to view the videos. Another capability would be to list all audio tracks by name that are used in the current world database, or the number of anchors that make use of a particular graphics image.

B. NON-TERRAIN ANCHORS

Currently Hyper-NPSNET utilizes only one kind of anchor. This is the Terrain anchor. Terrain anchors are fixed in 3D space and can only be changed through the anchor editor. Additional anchor types are planned and include vehicle anchors and temporal anchors.

1. Vehicle Anchors

A vehicle anchor is an anchor that is attached to a vehicle. The vehicle can move around in the virtual world with the anchor staying attached. Vehicle anchors are handy for visualizing vehicle capabilities or design. This kind of information is invaluable for individuals training on the simulator. For instance, a training tank driver may query an enemy vehicle he sees but can't identify. He can learn the type of vehicle, its locomotion and weapons capabilities, and even see a video informing him what is known of this vehicle from previous engagements on record. He could find out where the known weak points in the armor are and plan his strategy based on what he learns. When the soldier sees this vehicle again, he will be more capable to make the right decisions.

2. Temporal Anchors

Temporal anchors are useful when there is some time association of information that the user would like to explore. Temporal anchors would only exist over some time range and would contain information relevant to the time associated with the anchor. Such anchors would allow the visibility of attached information only during the window of time specified with the anchor. For example, a user of Hyper-NPSNET may only wish to view the video collected in March rather than have the display cluttered with the rest of the year's information temporal anchors.

C. NETWORKING

Hyper-NPSNET does not support any networking capability except minimal DIS to a sound server. The natural evolution of most software these days is toward having network abilities, and Hyper-NPSNET is no exception. The goal of Hyper-NPSNET is to supply hypermedia capability to our existing suite of battle field simulators known collectively as NPSNET [Zyda92]. A current goal of the NPSNET project is to construct simulators that are interoperable with the DARPA SIMNET system and the follow-on DIS networking standard [Institute91][Pope89].

D. TERRAIN DATABASE LOADING CAPABILITIES

Another shortcoming of Hyper-NPSNET is the limited support for a variety of terrains. Currently a 2 Km by 2 Km terrain database from Ft. Hunter Liggett in Central California is used. There is no current provision allowing other terrain databases to be loaded into Hyper-NPSNET. This improvement is a necessity in order to load any kind of virtual environment into the program. This also includes the objects that are normally thought of as being part of the terrain, like buildings, trees and rocks. As software for the generation of virtual environments advances, its clear that the terrain will need to play a more dynamic roll in the representation of features [2Zyda93]. Taking this a step further, the design of terrain base classes (in C++) will certainly contain stationary and non-stationary objects. This will address the question of what to render and what not to render. If a terrain object is determined to be in the field of view, then all objects that are “a part of” that terrain object will be rendered. The point is: in order to take advantage of the improvements in terrain design, Hyper-NPSNET will need the capability of loading terrain databases as a user command.

E. MORE SOPHISTICATED AUTHORING

As described in the section on authoring (see “AUTHORING WITH HYPER-NPSNET” on page 49), the authoring capabilities are somewhat limited. Individual anchors can be created and saved in hypermedia worlds. As the anchors are created, the user specifies the file names to be associated with the multimedia links of the information anchor. The user cannot easily view video clips, for instance, before assigning them to the anchor. Therefore, along the lines of the comments made above in Section A., the ability to view video and graphics files and listen to audio files during the authoring process would streamline the authoring and minimize the time spent in designing hypermedia documents.

F. USER INTERFACE DEVELOPEMENT

The user interface should undergo constant evolution. Since the user interface is the sole mechanism for the user to interact with Hyper-NPSNET, most of the above suggested improvements would be incorporated into the user interface. But beyond these specific improvements, as more users interact with the system, certain operations or situations will occur repeatedly and the user interface should change to make these operations more easily accomplished. The idea of improving the user interface is necessarily quite vague, because the interface is most of the program. This point is more a conceptual one than an implementation one, but certainly over time there will be additional capabilities in the system and the interface should change so as not to just add the new features but to incorporate them into an intelligent interface that is friendly and powerful.

G. STANDARDS COMPATIBILITY

A wealth of standards are emerging targeting multimedia and hypermedia data formats. Some standards are concerned with file data formats like JPEG or MPEG, while others are concerned with document structure like HyTime. An introduction to relevant standards is presented below. An important future capability of Hyper-NPSNET would be to read and write files and documents written in these new standards.

1. MHEG

The Multimedia Hypermedia Experts Group (MHEG) in the Joint Technical Committee 1 (JTC1) has joint participation from CCITT[1]. JTC1 is a combined effort from the International Standards Organization (ISO) and the International Electrotechnical Commission (IEC). The MHEG group is concerned with the coded representation of final form multimedia and hypermedia objects that will be interchanged across services and applications [Price93].

The MHEG group has listed some typical application domains, but in general anticipates the use of MHEG objects in large scale telematic applications: training and education, simulation and games, sales and advertising, office information systems, engineering documentation, culture, electronic publishing and electronic books, public

1. CCITT is a European Standards Committee.

information, computer supported cooperative work, medical applications and future classes of applications.

2. Hytime

HyTime is a SGML-based standard for the representation, archival storage, and interchange of multimedia and hypermedia documents [Newcomb91]. HyTime adds conventions to the SGML (Standard Generalized Markup Language, ISO 8879) which allows a variety of constructs, including hyperlinks and rendition instructions, to be expressed in a technology-neutral fashion. HyTime is being chosen as a “source code” representation by those who make large investments in the creation of hypertext and hypermedia documents, because it will protect such information from technological obsolescence.

3. JPEG

The Joint Photographic Experts Group (JPEG) is the informal name for ISO’s SC29/WG10. Through joint participation with CCITT Steering Group (SG) VIII, JPEG has developed a general purpose compression and encoding standard for grayscale and color “photographic” still images. This method allows compression and quality to be traded-off at compression time [Wallace91].

4. MPEG

The Moving Pictures Experts Group (MPEG) is the informal name for ISO’s SC29/WG11. Through joint participation with CCITT SG XV, MPEG has developed a standard (MPEG-1) for digital compression of VHS quality moving pictures and CD quality audio at around 1.5 Mbit/sec. A follow-on standard (MPEG-2), is applicable to compressed data rates from 3 to 15 Mbit/sec, with quality levels matching today’s laser video up through tomorrow’s HDTV [Le Gall91].

APPENDIX: HYPER-NPSNET USER MANUAL

Much of the detail of using Hyper-NPSNET has been described in various places through-out the body of this thesis. To simplify the learning process, a brief user manual is included here.

A. STARTING AND EXITING THE PROGRAM

Hyper-NPSNET is started at the command prompt by entering:

> hyper

The current directory must be `~lombardo/hyper/hyper` in order to access all the right subdirectories. This is not true if the user has designed his own hypermedia database, but is true if the `world.hyp` database, created by the author, is used.

To exit Hyper-NPSNET, use the **Exit** selection from the **File** pull-down menu or hit the **F3** key while the mouse cursor is anywhere on the main Hyper-NPSNET control panel.

B. LOADING AND SAVING HYPERMEDIA DATABASES

Existing hypermedia databases can be loaded using the **Open** selection from the **File** pull-down menu. Blank hypermedia databases can be established using the **New** selection from the **File** pull-down menu. To save a database under the same name as previously saved, use the **Save** selection from the **File** pull-down menu, and to save the database under a new name, use the **Save As** selection.

C. MOVEMENT THROUGH THE VIRTUAL WORLD

To move through the world, use the left and right mouse buttons. To move forward, use the left button and to move backwards, use the right button. When either button is pressed, a square red outline appears in the middle of the rendering window. Motion will be straight forward or backward, depending on which button was pressed, as long as the cursor is kept within the red square. To turn, merely move the cursor in the direction of the desired turn. Move the pointer left of the box to turn left and right of the box to turn right. If the pointer is moved above or below the box, the pitch can be changed to either climb or descend. The further the cursor is from the closest edge of the red square, the faster the rate of turning.

As motion is occurring, the speed is kept constant. The speed can be set or changed through the User Preference pop-up panel. The panel can be displayed using the **Preferences** selection from the **Edit** pull-down menu.

If the user becomes disoriented, the initial view can be reset using the **Reset View** command. This command is found on the **Display** pull-down menu on the main Hyper-NPSNET control panel.

D. TO CREATE OR EDIT INFORMATION ANCHORS

Creating new anchors and editing existing anchors are similar operations. All of this is done through the Anchor Editor Panel. The panel can be display using the **Anchor** selection from the **Edit** pull-down menu. With the Anchor Editor panel up, an anchor's name, type, orientation, coordinates, audio filename, video filename, graphics filename or text filename can be changed by placing the cursor in the text field widget and pressing the left mouse button to activate that text-field. Cutting and pasting work the same as in most other X based applications. When all changes are made, the anchor can be saved by selecting the **Save** button at the bottom of the panel. To revert back to the previously save version of the anchor, select the **Revert** button.

To create a new anchor, select the **New Anchor** button. All the text fields will be filled in with a default selection indicating this is a new anchor. Merely overwrite the entries with the desired values and then save the anchor. The new anchor will appear on the anchor list on the main panel. Once the **New Anchor** button is selected, the Anchor Editor goes into "New Anchor" mode. In this mode, new anchors can be rapidly added to the database. To exit this mode and go back to the "View Current Anchor" mode, select **Revert**. To close the anchor editor, select the **Close** button or select the **Anchor** command from the **Edit** pull-down menu again.

E. TO SET USER PREFERENCES

User preferences for Hyper-NPSNET include: flying speed, what anchors are displayed and when anchor information is displayed. These values are changed using the User Preferences Panel. This panel is displayed using the **Preferences** selection from the **Edit** pull-down menu.

To set the flying speed, move the cursor into the **Speed** text field widget and using a combination of highlighting with the mouse or using the delete key to erase characters, enter the desired speed. The default flying speed is 8.0. This is a relative number and not any absolute speed like meters/min.

To select that only local anchors are to be displayed, select the **Local Anchors Only** radio button. This will insure that only the anchors that are within the stated range from the users eye point will be displayed. The default distance is set to 300.0 meters. This distance can be changed to any value using the appropriate text field widget. The default is not to have Local Anchors Only, therefore all anchors will be displayed in the default setting.

Information attached to anchors can be retrieved automatically by selecting the **Anchor Auto View** radio button and the appropriate combination of what type of information desired. Once the automatic retrieval is set, whenever the user approaches within the specified distance from the anchor, the information is displayed. This is known as audio or video landmines. The default is no automatic retrieval.

F. ANCHOR SELECTION AND MULTIMEDIA QUERYING

Information anchors can be selected and queried in a few different ways. After loading a hypermedia database, all available anchors are listed in the scrolled listing widget on the main panel. Using the mouse, any anchor can be selected from the list by placing the mouse cursor over the desired anchor and either double clicking the left mouse button or pressing the left mouse button once followed by pressing the **Jump** button at the bottom of the listing widget. Upon selection in this manner, the user's eyepoint undergoes an instant aspect change to the coordinates and orientation of the selected anchor.

Another method for selecting anchors is to pick them right off the rendering window with the mouse. Position the mouse cursor on the desired anchor and press the middle mouse button. The selected anchor will be highlighted in the scrolled listing widget of the main panel. Selection of anchors in this way does not cause an aspect change for the user's eyepoint.

Either of the two methods described will cause the anchor selected to become the current selected anchor. Upon selection, the anchor name, type and coordinates will appear on the main Hyper-NPSNET panel. If the anchor has any multimedia files attached, then the appropriate **Audio**, **Video**, **Graphics** or **Text** buttons will be sensitive and can be pressed to view that file.

As described above, the user can set a preference to have certain anchor information displayed automatically as the user passes in close proximity to the anchor. This is **Anchor Auto View** mode, and is set using the User Preferences panel.

LIST OF REFERENCES

[Card91] Card, Stuart K., Robertson, George G., Mackinlay, Jock D., "The Information Visualizer: An Information Workspace," *Human Factors in Computing Systems* (ACM SIGCHI Conference Proceedings), 1991, pp. 181-188.

[Halasz88] Halasz, F.G., "Reflections on NoteCards: Seven Issues for the Next Generation of Hypermedia Systems," *Communications of the ACM*, Volume 31, Number 7, July 1988, pp. 836-852.

[Institute91] Institute for Simulation and Training, "Protocol Data Units for Entity Information and Entity Interaction in a Distributed Interactive Simulation," Military Standard (DRAFT), IST-PD-90-2, Orlando, FL, September 1991.

[Le Gall91] Le Gall, Didier, "MPEG: A Video Compression Standard for Multimedia Applications," *The Communications of the ACM*, Volume 34, Number 4, April 1991, pp. 46-58.

[Mackinlay91] Mackinlay, Jock D., Robertson, George G., Card, Stuart K., "The Perspective Wall: Detail and Context Smoothly Integrated," *Human Factors in Computing Systems* (ACM SIGCHI Conference Proceedings), 1991, pp. 173-179.

[Marcus92] Marcus, Aaron, *Graphic Design for Electronic Documents and User Interfaces,* ACM Press, Addison Wesley, 1992.

[de Mey93] de Mey, Vicky and Gibbs, Simon, "A Multimedia Component Kit, Experiences with Visual Composition of Applications," *ACM Multimedia 93 Conference Proceedings*, Anaheim, CA, August 1993, pp. 291-300.

[Newcomb91] Newcomb, Steven R., Kipp, Neill A., and Newcomb, Victoria T., "The HyTime Hypermedia/Time-based Document Structuring Language," *The Communications of the ACM*, Volume 34, Number 11, November 1991, pp. 67-83.

[Nielsen90] Nielsen, Jakob, *Hypertext and Hypermedia*, Academic Press, 1990.

[Phillips91] Phillips, Richard L., "MediaView: A General Multimedia Digital Publication System," *Communications of the ACM*, Volume 34, Number 7, July 1991, pp. 74-83.

[Pope89] Pope, Arthur, "The SIMNET Network and Protocols," BBN Report No. 7102, BBN Systems and Technologies, Cambridge, MA, July 1989.

[Price93] Price, Roger, “MHEG: An Introduction to the future International Standard for Hypermedia Object Interchange,” *ACM Multimedia 93 Conference Proceedings*, Anaheim, CA, August 1993, pp. 121-128.

[Robertson91] Robertson, George G., Mackinlay, Jock D., Card, Stuart K., “Cone Trees: Animated 3D Visualizations of Hierarchical Information,” *Human Factors in Computing Systems* (ACM SIGCHI Conference Proceedings), 1991, pp. 189-194.

[Wallace91] Wallace, Gregory K., “The JPEG Still Picture Compression Standard,” *The Communications of the ACM*, Volume 34, Number 4, April 1991, pp. 30-44.

[Ware90] Ware, Colin and Osborne, Steven, “Exploration and Virtual Camera Control in Virtual Three Dimensional Environments,” *1990 Symposium on Interactive 3D Graphics, ACM SIGGRAPH - Computer Graphics*, Volume 24, Number 2, March 1990, pp. 175-183.

[Wilson92] Wilson, Kalin P., “NPSGDL: An Object Oriented Graphics Description Language for Virtual World Application Support,” Masters’s Thesis, Naval Postgraduate School, Monterey, California, September 1992.

[Zyda91] Zyda, Michael and Pratt, David, “NPSNET: A 3D Visual Simulator for Virtual World Exploration and Experimentation,” *1991 SID International Symposium Digest of Technical Papers*, Volume XXII, 8 May 1991, pp. 361-364.

[Zyda92] Zyda, Michael J., Pratt, David R., Mohahan, James G., Wilson, Kalin P., “NPSNET: Constructing a 3D Virtual World,” *1992 Symposium on Interactive 3D Graphics*, March 1992, pp. 147-155.

[1Zyda93] Zyda, Michael J., Lombardo, Chuck, Pratt, David R., “Hypermedia and Networking in the Development of Large-Scale Virtual Environments,” *The Third International Conference on Artificial Reality and Tele-Existence”*, ICAT 93, July 6-7, 1993, pp. 33-39.

[2Zyda93] Zyda, Michael J., Pratt, David R., Falby, John S., Lombardo, Chuck and Kelleher, Kristen M., “The Software Required for the Computer Generation of Virtual Environments,” accepted for a future issue of *Presence.*

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
2.	Dudley Knox Library Code 052 Naval Postgraduate School Monterey, CA 93943	2
3.	Chairman, Code CS Computer Science Department Naval Postgraduate School Monterey, CA 93943	2
4.	Tradoc Analysis Command Code TRAC Naval Postgraduate School Monterey, CA 93943	2
5.	Professor Michael J. Zyda Code CSZk Computer Science Department Naval Postgraduate School Monterey, CA 93943	2
6.	LCDR John Falby Code CSFa Computer Science Department Naval Postgraduate School Monterey, CA 93943	2
7.	Professor David R Pratt Code CSPr Computer Science Department Naval Postgraduate School Monterey, CA 93943	1
8.	Mr. Charles P. Lombardo Computer Science Department Naval Postgraduate School Monterey, CA 93943	2